2019

Witness generation in existential CTL model checking

Chuan Jiang
Iowa State University

Witness generation in existential CTL model checking

by

Chuan Jiang

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
Gianfranco Ciardo, Major Professor
Andrew S. Miner
Samik Basu
Jack Lutz
Wensheng Zhang

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation. The Graduate College will ensure this dissertation is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2019

DEDICATION

I would like to dedicate this dissertation to my wife Wangyujue Hong without whose encouragement and support I would not have been able to complete this work. I would also like to thank my parents who supported me to study overseas and never left my side. In addition, I am very grateful to my friends, for many beautiful memories in Ames.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank the people who helped me with various aspects of conducting research and the writing of this dissertation. First and foremost, I would like to express my appreciation to my major professor, Dr. Gianfranco Ciardo, for his guidance, patience, and support throughout this research and the writing of this dissertation. I always received helpful and constructive suggestions and feedbacks from him. His insights and words of encouragement have often inspired me and renewed my hope for completing my graduate education. I would like to thank Dr. Andrew S. Miner for his suggestions in algorithm and experimental design. The collaborations with him were always efficient and pleasant. I would also like to thank the committee members, Dr. Samik Basu, Dr. Jack Lutz, and Dr. Wensheng Zhang, for their efforts and contributions to this work.

ABSTRACT

Hardware and software systems are widely used in applications where failure is prohibitively costly or even unacceptable. The main obstacle to make such systems more reliable and capable of more complex and sensitive tasks is our limited ability to design and implement them with sufficiently high degree of confidence in their correctness under all circumstances. As an automated technique that verifies the system early in the design phase, model checking explores the state space of the system exhaustively and rigorously to determine if the system satisfies the specifications and detect fatal errors that may be missed by simulation and testing. One essential advantage of model checking is the capability to generate witnesses and counterexamples. They are simple and straightforward forms to prove an existential specification or falsify a universal specification. Beside enhancing the credibility of the model checker's conclusion, they either strengthen engineers' confidence in the system or provide hints to reveal potential defects.

In this dissertation, we focus on symbolic model checking with specifications expressed in computation tree logic (CTL), which describes branching-time behaviors of the system, and investigate the witness generation techniques for the existential fragment of CTL, i.e., ECTL, covering both decision-diagram-based and SAT-based.

Since witnesses provide important debugging information and may be inspected by engineers, smaller ones are always preferable to ease their interpretation and understanding. To the best of our knowledge, no existing witness generation technique guarantees the minimality for a general ECTL formula with nested existential CTL operators. One contribution of this dissertation is to fill this gap with the minimality guarantee. With the help of the saturation algorithm, our approach computes the minimum witness size for the given ECTL formula in every state, stored as an additive edge-valued multiway decision diagrams ($EV^{+}MDD$), a variant of the well-known binary decision

diagram (BDD), and then builds a minimum witness. Though computationally intensive, this has promising applications in reducing engineers' workload.

SAT-based model checking, in particular, bounded model checking, reduces a model checking problem problem into a satisfiability problem and leverages a SAT solver to solve it. Another contribution of this dissertation is to improve the translation of bounded semantics of ECTL into propositional formulas. By realizing the possibility of path reuse, i.e., a state may build its own witness by reusing its successor's, we may generate a significantly smaller formula, which is often easier for a SAT solver to answer, and thus boost the performance of bounded model checking.

CHAPTER 1. OVERVIEW

Today, hardware and software systems are widely used in applications where failure is prohibitively costly or even unacceptable, such as e-commerce, medical instruments, highway and air traffic control systems, autonomous cars, and space exploration. A recent example of failure caused by errors in hardware or software systems is the Hitomi satellite, which launched on February 17, 2016 and broke up into pieces after five weeks, due to a series of cascading incidents [92]. The brightness threshold for stars to be noticed was set too high such that the star tracker could not find enough stars to do a proper orientation. Hitomi relied instead on the gyroscopes to calculate its orientation. But the gyroscopes were reporting, erroneously, that Hitomi was rotating. The reaction to counter the rotation put the satellite into an actual spin, which got faster and faster. The onboard rocket thrusters could have realigned and saved the satellite, but the instructions uploaded to them turned out to be incorrect, which could have been caught if the instructions had been tested on a computer simulation. This $273 million satellite had been seen as the future of X-ray astronomy, and its breakup was thought as a scientific tragedy.

Undoubtedly, designing and building reliable hardware and software systems are critical. The main obstacle to make such systems more reliable and capable of more complex and sensitive tasks is not inadequate speed or computation power of the existing machines, but our limited ability to design and implement them with sufficiently high degree of confidence in their correctness under all circumstances. The typical development process of hardware and software systems is illustrated in Figure 1.1. *Design verification*, which aims to ensure the correctness of the design at the earliest stage possible, is an essential step conducted in the design phase of any responsible system development process.

Simulation and *testing* are widely-adopted verification techniques in practice. They both involve running experiments before system deployment. Simulation is performed on an abstraction or a

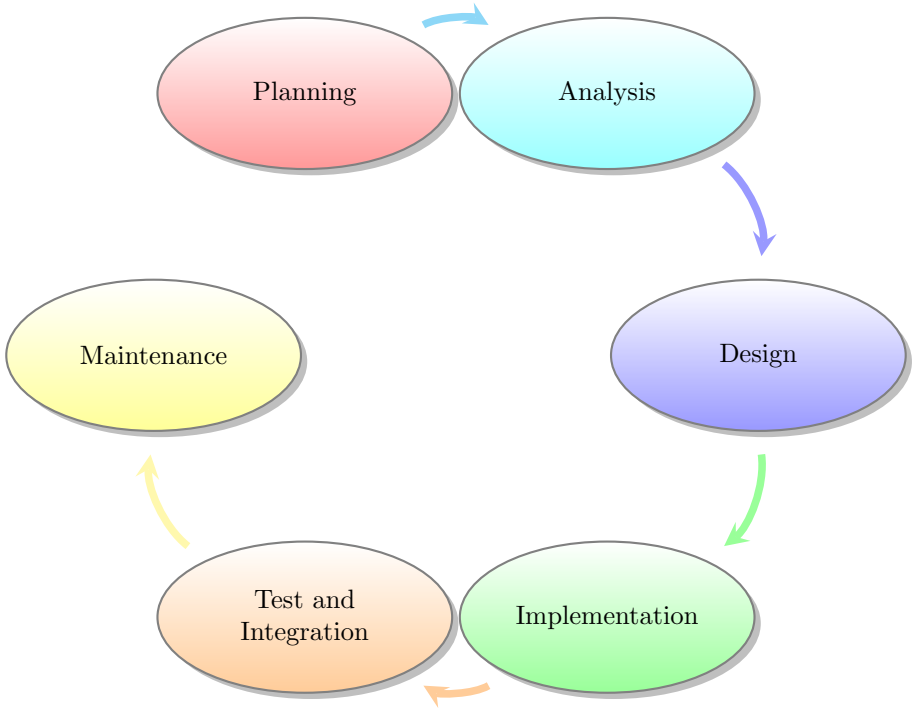


Figure 1.1: Development process of hardware and software systems

model of the system in the design phase, while testing is performed on the actual system in the implementation and integration phase. In both cases, they typically involve providing certain inputs and observing whether the corresponding outputs meet the expectations. These methods can be a cost-efficient way to find many errors in the very early stage of debugging, when the design or the system is still infested with multiple bugs. Their effectiveness drops quickly as the design becomes cleaner or the system becomes more mature. They can also easily miss significant errors when the number of possible states in the system is very large.

While simulation and testing explore some of possible behaviors and scenarios of the system, leaving open the question of whether the unexplored areas contain fatal errors, an attractive alternative is the approach of *formal verification*, which conducts an exhaustive exploration of all possible behaviors. When a formal verification technique concludes that a design is correct, it implies that all the behaviors have been explored and verified, without the problem of inadequate coverage.

Deductive verification is one of the key approaches of formal verification. It usually involves generating from the system and its specifications a collection of mathematical obligations, the truth of which imply the conformance of the system to its specifications, and constructing proofs for them using mathematical axioms and proof rules. Initially, these proofs were constructed by hand. Eventually, software tools, such as theorem provers (e.g. Isabelle [73], PVS [72], and HOL [42]), were developed to apply a systematic search to suggest various ways to progress from the current stage of the proof. This approach is typically time consuming and requires a great amount of manual intervention. Users must have expertise in logical reasoning and considerable experience to understand why the system works correctly and convey this information to the verification system, in the form of either a sequence of theorems to be proved or specifications of system components.

Another formal verification approach is *model checking* [33], on which the main topic of this dissertation is based. It is an automatic technique for verifying finite state concurrent systems, which arise naturally in several areas of computer science, especially in the design of digital circuits and communication protocols. It is also applicable for some infinite systems where infinite sets of states can be represented finitely by using abstraction or taking advantage of symmetry. In this approach, specifications are expressed over state transition systems in temporal logic such as *linear time logic* (LTL) and *computation tree logic* (CTL). An exhaustive and efficient search procedure is used to explore the state space of the system and determine if the system satisfies the specification.

Compared to deductive verification, the approach of model checking enjoys two remarkable advantages:

- Model checking is fully automatic, and its application does not require user supervision or expertise in mathematical disciplines such as logic reasoning and theorem proving. Users who can run simulations of a design are fully qualified and capable of model checking the same design.

- When a model checker concludes that a system satisfies or fails to satisfy a desired property, it is able to produce a witness or counterexample to demonstrate such satisfaction or violation. Witnesses are particularly useful for engineers to understand system behaviors, and

counterexamples provide a priceless insight to understanding the real reason for the failure as well as important clues for fixing the problem.

The major challenge of model checking is dealing with the *state explosion* problem, which occurs if the system has many components that interact with each other and evolve concurrently. In such cases the number of global system states can be enormous and grows exponentially with the number of components. Much of the research in model checking over the past 30 years has involved developing techniques for tackling this problem. Based on how system states are represented, the methodologies of model checking can be divided into either *explicit* or *symbolic* (or *implicit*). *Explicit model checking* enumerates states individually, storing them in structures like hash tables. It tends to present relatively more predictable time and memory efficiency. However, the complexity is linear in the number of states in the system, thus the explicit approach does not scale well for complex systems. On the other hand, *symbolic model checking* uses efficient encoding of logical formulas to represent and to manipulate sets of states and transitions simultaneously. It avoids explicitly constructing the graph for finite state systems and allows us to verify systems clearly out of reach of the explicit approach. Since the late 1980s, the size of systems that could be verified by model checking techniques has increased dramatically, mainly due to the introduction of the symbolic approach.

This dissertation focuses on improving the witness generation techniques for CTL symbolic model checking, covering both decision-diagram-based and SAT-based. Logically, the negation of an existential statement is a universal statement, and vice versa. In general, it is only feasible and useful to find a witness to demonstrate that an existential statement is true, and a counterexample to demonstrate that a universal statement is false. Therefore, witness and counterexamples are essentially the same concepts, and the approaches and discussions in the dissertation are also applicable for counterexample generation. Compared to the proofs provided by theorem provers, witnesses and counterexamples, usually in the form of actual execution traces, are most of time easier and more straightforward to understand. Engineers can simply track the transitions of system states to gain a clear picture of expected or unexpected system behaviors. Experience has shown

that witness and counterexample generation are effective features convincing engineers of the value of model checking.

The rest of the dissertation is organized as follows. Chapter 2 provides an overview of system representations, computation tree logic (CTL), witnesses and counterexamples. Chapter 3 introduces symbolic CTL model checking based on decision diagrams and proposes an approach that generates the minimum witness using edge-valued decision diagrams. Chapter 4 introduces SAT-based symbolic model checking, more precisely, bounded model checking, and proposes an improved encoding approach for bounded CTL semantics. Experimental results and future work are also presented in Chapter 3 and Chapter 4. Chapter 5 concludes this dissertation.

CHAPTER 2. BACKGROUND

Throughout this dissertation, we denote sets using calligraphic letters (e.g., $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$), except for the booleans $\mathbb{B} = \{\mathbf{0}, \mathbf{1}\}$ or $\{\mathbf{true}, \mathbf{false}\}$, the natural numbers $\mathbb{N} = \{0, 1, 2, ...\}$, and $\mathbb{N}_\infty = \mathbb{N} \bigcup \{\infty\}$.

2.1 System Representations

The first task of model checking is to convert a design into a formalism accepted by a model checker. Due to limitations on time and memory, the modeling of a design may require abstraction to eliminate irrelevant or unimportant details. We introduce two kinds of system representations. Section 2.1.1 presents Kripke structures, which are commonly used to describe state spaces in the literature. Section 2.1.2 presents Petri nets, which are especially suitable for modeling concurrent, asynchronous, and nondeterministic systems. Although these formalisms are very simple, they are sufficiently expressive to capture aspects of temporal behavior that are most important for reasoning about hardware and software systems.

2.1.1 Kripke Structures

Kripke structures were originally proposed by Saul Kripke [60]. As a variant of a transition system, a Kripke structure is basically a graph whose nodes represent states of the system and edges represent state transitions. Formally, a Kripke structure is a tuple $(\mathcal{S}, \mathcal{S}_{init}, \mathcal{N}, \mathcal{A}, \mathcal{L})$, where:

- $\mathcal{S}$ is a finite state space.
- $\mathcal{S}_{init} \subseteq \mathcal{S}$ is the set of initial states.
- $\mathcal{N} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation, which must be left total, i.e., for every state $s \in S$, there is a state $s' \in S$ such that $(s, s') \in \mathcal{N}$.
- $\mathcal{A}$ is a set of atomic propositions.

- $\mathcal{L} : \mathcal{S} \to 2^{\mathcal{A}}$ is a labeling function that gives the atomic propositions holding in each state (subject to **true** $\in \mathcal{A}$ holding in every state).

$\mathcal{N}$ can be analogously viewed as a *next-state function* $\mathcal{N} : \mathcal{S} \to 2^{\mathcal{S}}$ such that $\mathcal{N}(s)$ is the set of states that can be nondeterministically reached in one step from state s: $\mathcal{N}(s) = \{s' \mid (s, s') \in \mathcal{N}\}$. We use such set-typed and function-typed representations indistinguishably and interchangeably. The inverse transition relation $\mathcal{N}^{-1}$ is obtained by swapping the two states in every transition in $\mathcal{N}$. $\mathcal{N}^{-1}$ can also be analogously viewed as a *previous-state function* $\mathcal{N}^{-1} : \mathcal{S} \to 2^{\mathcal{S}}$ such that $\mathcal{N}^{-1}(s)$ is the set of states that can nondeterministically reach state s in one step: $\mathcal{N}^{-1}(s) = \{s' \mid (s, s') \in \mathcal{N}^{-1}\} = \{s' \mid (s', s) \in \mathcal{N}\}$.

Figure 2.1 illustrates a simple Kripke structure $(\mathcal{S}, \mathcal{S}_{init}, \mathcal{N}, \mathcal{A}, \mathcal{L})$, where $\mathcal{S} = \{s_0, s_1, s_2\}$, $\mathcal{S}_{init} = \{s_0\}$, $\mathcal{N} = \{(s_0, s_1), (s_1, s_0), (s_0, s_2), (s_1, s_2), (s_2, s_2)\}$, and the atomic propositions a and b hold in s_0 and s_2, respectively. States pointed by edges without source nodes are initial states. Absence of atomic propositions aside a state implies their negations hold in that state. For example, $\neg a$ holds in s_1 and s_2, while $\neg b$ holds in s_0 and s_1.

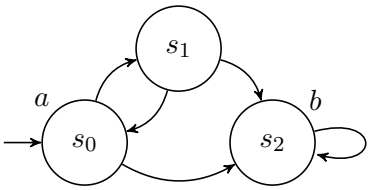


Figure 2.1: A Kripke structure $(\mathcal{S}, \mathcal{S}_{init}, \mathcal{N}, \mathcal{A}, \mathcal{L})$.

A *path* starting from a state s is an infinite sequence of states $\pi = [\![s_0, s_1, s_2, ...]\!]$ such that $s_0 \equiv s$ and $(s_i, s_{i+1}) \in \mathcal{N}$ for all $i \in \mathbb{N}$ (see Figure 2.2(a)). A *finite path* is a finite prefix of a path. Since $\mathcal{N}$ is left total, i.e., every state has at least one successor, every finite path can be extended into an infinite path. Let $Path(s)$ be the set of paths starting at s. A *cycle* is a path that can be represented with its finite prefix $[\![s_0, s_1, ..., s_n]\!]$ where $n \geq 1$ and $s_0 \equiv s_n \equiv s$ (see Figure 2.2(b)). Let $Cycle(s) \subseteq Path(s)$ be the set of cycles starting at s.

(a) A path $[\![s_0, s_1, s_2, ...]\!]$ (b) A cycle $[\![s_0, s_1, ..., s_{n-1}, s_n \equiv s_0]\!]$

Figure 2.2: Path and cycle.

2.1.2 Petri Nets

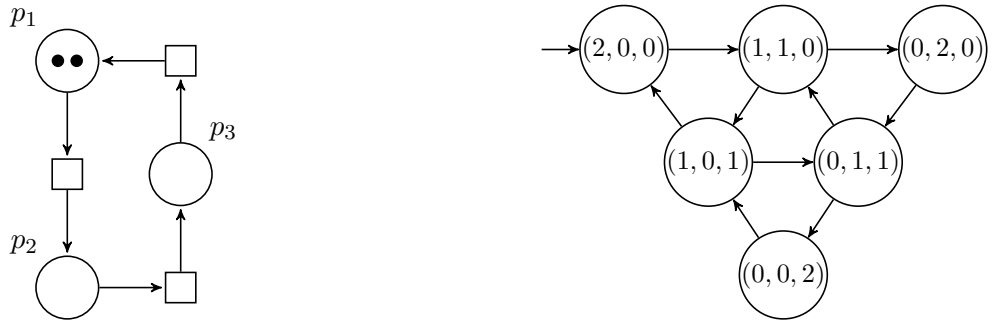
Petri nets, also known as *place/transition nets*, were invented by Carl Adam Petri [75] for the purpose of describing chemical processes. A Petri net is a directed bipartite graph whose nodes are divided into places and transitions. Formally, a Petri net is a tuple $(\mathcal{P}, \mathcal{T}, D^-, D^+, \mathbf{m}_{init})$, where:

- $\mathcal{P}$ is a set of *places*, drawn as circles.
- $\mathcal{T}$ is a set of *transitions*, drawn as rectangles.
- $D^- : \mathcal{P} \times \mathcal{T} \to \mathbb{N}$ and $D^+ : \mathcal{P} \times \mathcal{T} \to \mathbb{N}$ are the *input arc* and the *output arc* cardinalities, respectively.
- $\mathbf{m}_{init} \in \mathbb{N}^{|\mathcal{P}|}$ is the *initial marking*, specifying a number of *tokens* initially present in each place.

A *state* of the Petri net is determined by the *marking function* $\mathbf{m} : \mathcal{P} \to \mathbb{N}$ that gives the number of tokens in each place. A Petri net is a high-level representation of a special discrete-state system with a potential state space $\mathcal{S}_{pot} = \mathbb{N}^{|\mathcal{P}|}$, an initial state set $\mathcal{S}_{init} = \{\mathbf{m}_{init}\}$, and a transition relation $\mathcal{N} \subseteq \mathcal{S}_{pot} \times \mathcal{S}_{pot}$ given by the union $\mathcal{N} = \bigcup_{\alpha \in \mathcal{T}} \mathcal{N}_\alpha$ of the transition relation for each Petri net transition α.

Petri nets are often used to describe concurrent, asynchronous, and nondeterministic systems consisting of components and can represent systems with infinite number of states. Throughout this dissertation, we assume that the Petri net is *bounded*, meaning that the number of tokens in every place is below a certain value in every reachable state and thus $\mathcal{S}_{pot}$ contains a finite number of states. Furthermore, a Petri net is *safe* if each place contains at most one token (therefore, $\mathcal{S}_{pot} = \mathbb{B}^{|\mathcal{P}|}$).

If a Petri net is bounded and the transition relation of the corresponding discrete-state system is left total, it can be represented as a Kripke structure, since a Kripke structure is also a special discrete-state system. The atomic propositions in the Kripke structure are relations interpreted over the markings of the Petri net. An example is presented in Figure 2.3, where a state consists of the numbers of tokens in $p1$, $p2$, and $p3$.



(a) A Petri net (b) The corresponding Kripke structure

Figure 2.3: A Petri net and the corresponding Kripke structure.

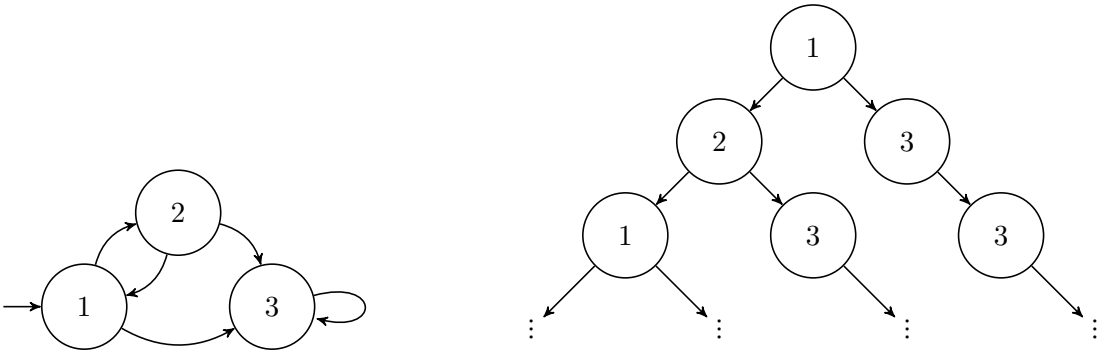
In the remainder of the dissertation, we always describe and explain with Kripke structures as examples for their simplicity, while Petri nets are used in the experiments for their conciseness when describing large state spaces.

2.2 Computation Tree Logic

Before verification, it is necessary to describe the specifications that the system must conform to. These specifications are often given unambiguously in some logical formalism. *Temporal logics* have proved to be useful for describing sequences of transitions between states in concurrent systems, since they can specify the ordering of events without introducing time explicitly. They are often classified according to whether time is assumed to have a *linear* or a *branching* structure.

In this dissertation, we concentrate on *computation tree logic* (CTL), which was introduced by Clark and Emerson [31, 39] in the early 1980s to describe branching-time temporal behaviors over *computation trees*. A computation tree is formed by designating a state in a Kripke structure as the initial state and then unwinding the structure into an infinite directed tree with the initial state

at the root, as shown in Figure 2.4. The tree illustrates all the possible executions starting from the initial state.



(a) A Kripke structure (b) The corresponding computation tree

Figure 2.4: A Kripke structure and its computation tree.

Every CTL operator is composed of a *path quantifier* and a *temporal operator*. The path quantifiers describe the branching structure in the computation tree. A ("*for all computation paths*") and E ("*for some computation paths*") are two such quantifiers, which are used in a particular state to specify that all the paths or some of the paths starting from that state satisfy some property. The temporal operators describe properties of a path through the computation tree. The four basic temporal operators are as follows:

- The X ("*next time*") operator requires that the second state on the path satisfies a property.
- The F ("*eventually*" or "*in the future*") operator requires some state on the path satisfies a property.
- The G ("*globally*") operator requires every state on the path satisfies a property.
- The U ("*until*") operator combines two properties and is a little more complicated. It requires that some state on the path satisfies the second property, and every preceding state on the path satisfies the first property.

The two path quantifiers and the four temporal operators form the eight CTL operators EX, EF, EG, EU, AX, AF, AG, and AU. Among them, EX, EF, EG and EU are *existential* operators, and AX, AF, AG and AU are *universal* operators.

With logical connectives and CTL operators, the syntax of CTL formulas is defined as follows:

$$\begin{aligned}\varphi \ ::= \ & a \mid \neg\varphi \mid \varphi \wedge \rho \mid \varphi \vee \rho \mid \\ & \mathsf{EX}\varphi \mid \mathsf{EF}\varphi \mid \mathsf{EG}\varphi \mid \mathsf{E}(\varphi \mathsf{U} \rho) \mid \\ & \mathsf{AX}\varphi \mid \mathsf{AF}\varphi \mid \mathsf{AG}\varphi \mid \mathsf{A}(\varphi \mathsf{U} \rho)\end{aligned}$$

where $a \in \mathcal{A}$ is an atomic proposition.

Now, we define the semantics of CTL formally. Given a Kripke structure $M = (\mathcal{S}, \mathcal{S}_{init}, \mathcal{N}, \mathcal{A}, \mathcal{L})$, the conditions for state $s \in S$ to satisfy CTL formula φ, written $M, s \models \varphi$ (M is omitted if it implicitly understood), are defined as follows:

$$\begin{aligned}
s \models a &\Leftrightarrow a \in \mathcal{L}(s) \\
s \models \neg\varphi &\Leftrightarrow s \not\models \varphi \\
s \models \varphi \wedge \rho &\Leftrightarrow s \models \varphi \text{ and } s \models \rho \\
s \models \varphi \vee \rho &\Leftrightarrow s \models \varphi \text{ or } s \models \rho \\
s \models \mathsf{EX}\varphi &\Leftrightarrow \exists [\![s_0, s_1, ...]\!] \in Path(s), s_1 \models \varphi \\
s \models \mathsf{EF}\varphi &\Leftrightarrow \exists [\![s_0, s_1, ...]\!] \in Path(s), \exists i \geq 0, s_i \models \varphi \\
s \models \mathsf{E}(\varphi \mathsf{U} \rho) &\Leftrightarrow \exists [\![s_0, s_1, ...]\!] \in Path(s), \exists i \geq 0, s_i \models \rho \wedge \forall j \in \{0, ..., i-1\}, s_j \models \varphi \\
s \models \mathsf{EG}\varphi &\Leftrightarrow \exists [\![s_0, s_1, ...]\!] \in Path(s), \forall i \geq 0, s_i \models \varphi \\
s \models \mathsf{AX}\varphi &\Leftrightarrow \forall [\![s_0, s_1, ...]\!] \in Path(s), s_1 \models \varphi \\
s \models \mathsf{AF}\varphi &\Leftrightarrow \forall [\![s_0, s_1, ...]\!] \in Path(s), \exists i \geq 0, s_i \models \rho \\
s \models \mathsf{A}(\varphi \mathsf{U} \rho) &\Leftrightarrow \forall [\![s_0, s_1, ...]\!] \in Path(s), \exists i \geq 0, s_i \models \rho \wedge \forall j \in \{0, ..., i-1\}, s_j \models \varphi \\
s \models \mathsf{AG}\varphi &\Leftrightarrow \forall [\![s_0, s_1, ...]\!] \in Path(s), \forall i \geq 0, s_i \models \varphi
\end{aligned}$$

Definition 2.2.1. *A CTL formula φ is universally valid in a Kripke structure $M = (\mathcal{S}, \mathcal{S}_{init}, \mathcal{N}, \mathcal{A}, \mathcal{L})$, written $M \models_A \varphi$, if and only if for every $s \in \mathcal{S}_{init}$, $s \models \varphi$.*

Definition 2.2.2. *A CTL formula φ is existentially valid in a Kripke structure $M = (\mathcal{S}, \mathcal{S}_{init}, \mathcal{N}, \mathcal{A}, \mathcal{L})$, written $M \models_E \varphi$, if and only if for some $s \in \mathcal{S}_{init}$, $s \models \varphi$.*

When S_{init} contains only one state, universal and existential validity coincide. Without loss of generality, in the rest of the dissertation, we assume that every system has a single initial state and write $M \models \varphi$ for validity. If φ is valid in M, M is said to be a *model* of φ.

Consider a state $s \in \mathcal{S}$. $\mathsf{EX}\varphi$ holds in s if and only if s has a successor s' such that φ holds in s'. $\mathsf{EF}\varphi$ holds in s if and only if there exists a path $\pi \in Path(s)$ containing a state s' such that φ holds in s'. $\mathsf{E}(\varphi \mathsf{U} \rho)$ holds in s if and only if there exists a path $\pi \in Path(s)$ containing a state s' such that ρ holds in s' and φ holds in every state before s' along π. $\mathsf{EG}\varphi$ holds in s if and only if there exists a path $\pi \in Path(s)$ such that φ holds in every state along π. AX, AF, AU, and AG are the corresponding universal operators over all the paths in $Path(s)$. The eight CTL operators are illustrated in Figure 2.5, where state s is the root of each computation tree.

According to the definitions, AX, AF, and AG are dual to EX, EG, and EF, respectively. In fact, EX, EU, and EG are adequate to express all the CTL operators:

$$s \models \mathsf{EF}\varphi \Leftrightarrow s \models \mathsf{E}(\mathbf{true}\mathsf{U}\varphi)$$

$$s \models \mathsf{AX}\varphi \Leftrightarrow s \not\models \mathsf{EX}\neg\varphi$$

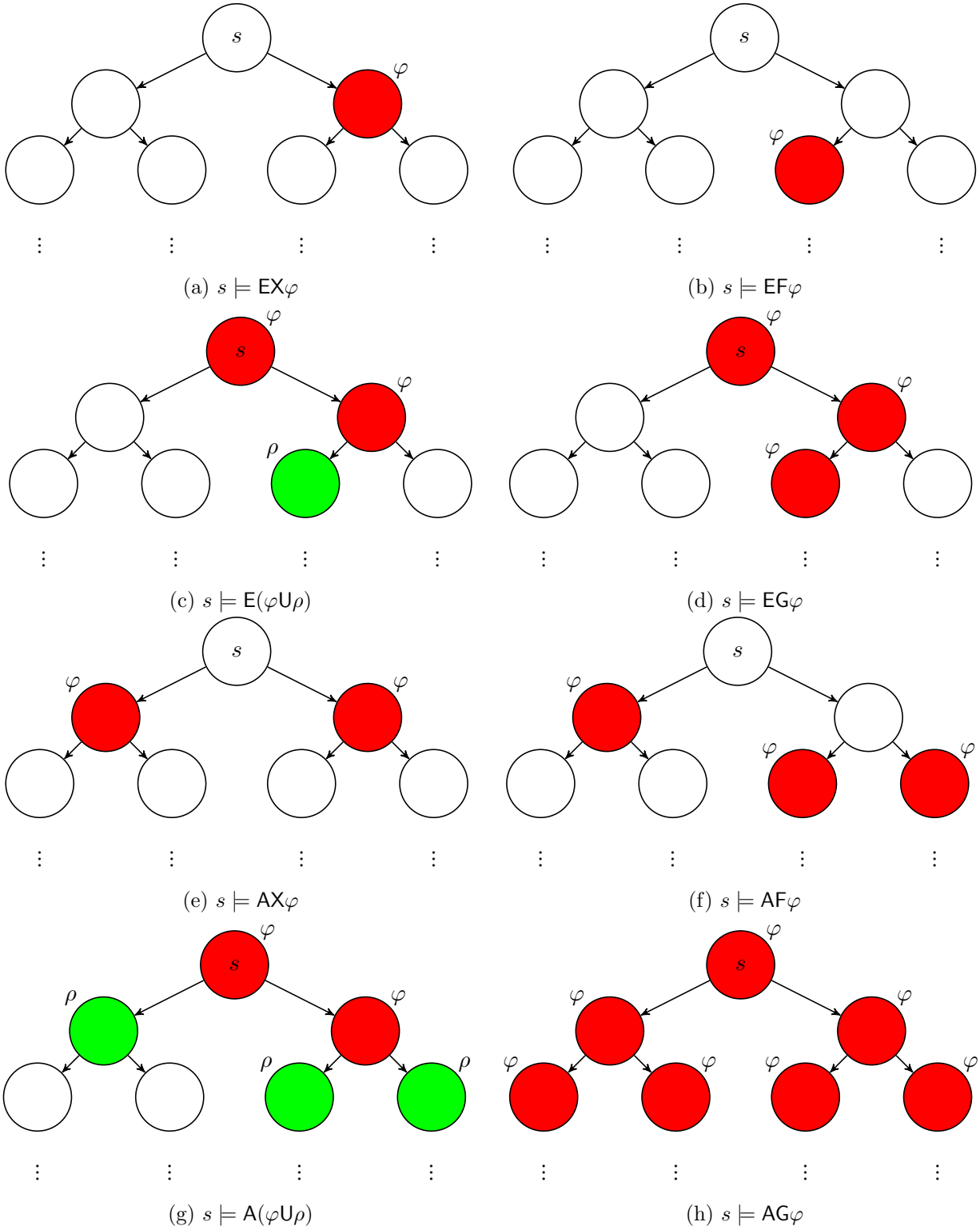
$$s \models \mathsf{AF}\varphi \Leftrightarrow s \not\models \mathsf{EG}\neg\varphi$$

$$s \models \mathsf{AG}\varphi \Leftrightarrow s \not\models \mathsf{EF}\neg\varphi$$

$$s \models \mathsf{A}(\varphi \mathsf{U} \rho) \Leftrightarrow s \not\models \mathsf{EG}\neg\rho \vee \mathsf{E}(\neg\rho \mathsf{U}(\neg\varphi \wedge \neg\rho))$$

Therefore, most CTL model checkers implement the algorithms for EX, EU, and EG formulas only. The other CTL operators are implemented based on them.

In order to avoid implicit existential or universal temporal operators resulting from the use of negation, we write and assume the formulas in *negation normal form* (NNF), i.e., negations are only applied to atomic propositions. Then, *ECTL* is the existential fragment of CTL where the only applicable CTL operators are existential ones (e.g., $\mathsf{EG}(\mathsf{EF}\varphi)$ and $\mathsf{E}((\mathsf{EX}\varphi)\mathsf{U}\rho)$). Similarly, *ACTL* is the universal fragment of CTL where the only applicable CTL operators are universal ones (e.g.,



(a) $s \models \mathsf{EX}\varphi$

(b) $s \models \mathsf{EF}\varphi$

(c) $s \models \mathsf{E}(\varphi \mathsf{U} \rho)$

(d) $s \models \mathsf{EG}\varphi$

(e) $s \models \mathsf{AX}\varphi$

(f) $s \models \mathsf{AF}\varphi$

(g) $s \models \mathsf{A}(\varphi \mathsf{U} \rho)$

(h) $s \models \mathsf{AG}\varphi$

Figure 2.5: The eight CTL operators.

$\mathsf{AG}(\mathsf{AF}\varphi)$ and $\mathsf{A}((\mathsf{AX}\varphi)\mathsf{U}\rho))$. The intersection of ECTL and ACTL falls into propositional logic (PL). Their relations are depicted in Figure 2.6.

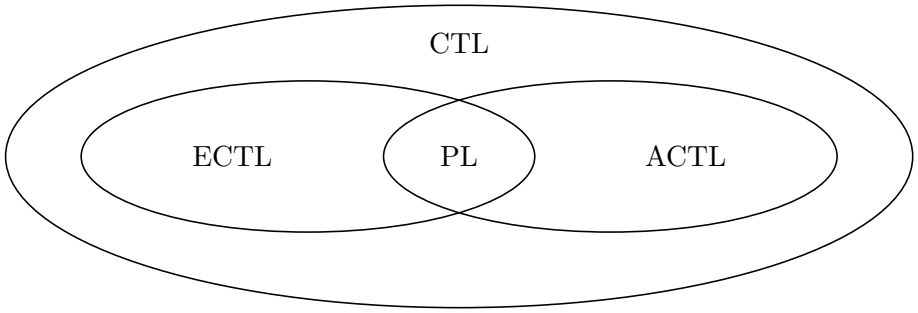


Figure 2.6: CTL, ECTL, ACTL and PL.

2.2.1 Witnesses and Counterexamples

When a model checker finishes verifying a specification φ over a Kripke structure M, since the state space of M can be very large, we expect the model checker to provide a *witness* if M is determined to satisfy φ, or a *counterexample* if M is determined to violate φ. A *witness* W is a substructure of M such that $W \models \varphi$ and the satisfaction of φ on W "explains" the satisfaction on M in a rigorous manner. Likewise, a *counterexample* C is a substructure of M such that $C \not\models \varphi$ and the violation of φ on C "explains" the violation of φ on M in a rigorous manner. Besides enhancing the credibility of the model checker's conclusion, they either strengthen engineers' confidence in the system or provide hints to reveal potential defects.

Using $W = M$ as a witness and $C = M$ as a counterexample, while simple and mathematically correct, is just a simple-minded choice. It is natural to seek W or C which are simple enough to be practically useful, but rich enough to exhibit all behaviors that are relevant to satisfaction or violation of the specification. Clark et al. [27, 30] identify the following criteria for good classes $\mathcal{C}$ of counterexamples (which are also applicable to witnesses):

- **Completeness:** $\mathcal{C}$ should be complete for a tangible class of specifications, i.e., each violation of a specification is witnessed by a suitable counterexample in $\mathcal{C}$.

- **Intelligibility:** The counterexamples in $\mathcal{C}$ should be simple and specific enough to be analyzed by human engineers, possibly with the aid of automated tools and suitable annotations.
- **Uniformity:** The counterexamples in $\mathcal{C}$ should be related to the specification and the system by a uniform principle which explains the violation.
- **Effectiveness:** There should be efficient algorithms for generating and manipulating counterexamples; in particular, computation of counterexamples for a specification φ should not be harder than the model checking problem for φ.

There is no formal definition of classes of counterexamples that fulfill these criteria, since notions such as intelligibility cannot be captures mathematically.

In the context of CTL, we generate witnesses only for ECTL and counterexamples only for ACTL, because we can expect to have simple natural witnesses only for existential specifications, and simple natural counterexamples only for universal specifications. Though most of the literature investigates counterexamples for ACTL, the conclusions from it are also applicable to witnesses for ECTL, since a witness to an ECTL formula is the dual of a counterexample to an ACTL formula.

The shapes of witnesses vary by models and temporal logics. Linear witnesses are simply in the form of paths. While counterexamples for LTL formulas, if they exist, are always linear (we only consider counterexamples for LTL because LTL formulas describe universal specifications), the situation is more complicated when considering CTL. For example, a witness for $s \models \mathsf{E}((\mathsf{EF}a)\mathsf{U}b)$ is likely to be constituted by a set of paths: a path $[\![s_0, s_1, ..., s_n, ...]\!]$ where $s_0 \equiv s$ and $s_n \models b$, and n paths such that for every $i \in \{0, ..., n-1\}$, there is a path demonstrating that a state satisfying a can be reached from s_i. When fully unfolded, this witness is actually a infinite computation tree whose root node is s. See Figure 2.7 for one such witness. Therefore, linear witnesses are not complete for ECTL. Likewise, linear counterexamples are not complete for ACTL.

Though fully unfolded witnesses and counterexamples are infinite paths or computation trees by definition, their finite prefixes are sufficient to explain the satisfaction or violation. Since the transition relation of a Kripke structure is left total and a state may have multiple successors, every

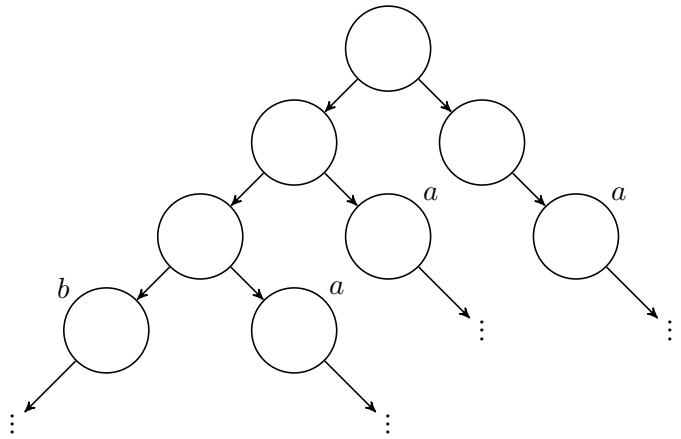


Figure 2.7: A fully unfolded witness for $\mathsf{E}((\mathsf{EF}a)\mathsf{U}b)$.

finite prefix can be extended into a set of infinite structures. A finite path or computation tree actually represents a set of witnesses or counterexamples. In this dissertation, we assume such finite representation to study witnesses and counterexamples.

2.2.1.1 Linear Counterexamples and Witnesses

Since linear counterexamples provide an easy and intuitive means to study system behavior, efforts have been made to identify ACTL formulas with linear counterexamples. Linear counterexamples are closely related to the linear fragment of ACTL. Consequently, identifying ACTL formulas with linear counterexamples amounts to investigating the linear fragment of ACTL. Maidi characterized this fragment as ACTL^{det} [64], defined as follows:

> If φ_1 and φ_2 are in ACTL^{det}, and p is an atomic formula, then p, $\varphi_1 \wedge \varphi_2$, $(p \wedge \varphi_1) \vee (\neg p \wedge \varphi_2)$, $\mathsf{AX}\varphi_1$, $\mathsf{A}((p \wedge \varphi_1)\mathsf{U}(\neg p \wedge \varphi_2))$, $\mathsf{A}((p \wedge \varphi_1)\mathsf{W}(\neg p \wedge \varphi_2))$ are in ACTL^{det}.

The W ("*weak until*") operator requires that every state on the path satisfies the first property until a state on the path satisfies the second property, but it does not require such state exists, i.e., $s \models \varphi\mathsf{W}\rho \Leftrightarrow s \models G\varphi \vee (\varphi\mathsf{U}\rho)$. She also showed that it is PSPACE-complete to determine if an ACTL formula is equivalent to a formula in ACTL^{det}. Therefore, Clark et al. concluded that computing a linear counterexample is a hard problem [30]. Unless P = PSPACE, for every polynomial CTL model checker which generates linear counterexamples, there exist infinitely many cases where the

linear counterexample produced by the model checker is not complete. Moreover, the fragment of ACTL which guarantees the existence of a linear counterexample cannot be captured by simple syntactic means, as the decision procedure is PSPACE-complete.

Buccafurri et al. introduced the concept of ACTL templates [15] to capture ACTL formulas which guarantee linear counterexamples whenever they do not hold. An ACTL template is an ACTL formula where $*$ is the single atomic proposition used. An instantiation of a template is obtained by replacing each occurrence of $*$ in the template by a pure propositional formula. There exists a unique maximal set of ACTL templates whose instantiations guarantee linear counterexamples. This set is given by a context-free grammar LIN in BNF notation:

$$\begin{aligned} LIN &::= PSF \mid LIN \wedge LIN \mid LIN \vee PSF \mid PSF \vee LIN \mid \mathsf{AX}(LIN) \mid \mathsf{A}(PSF \mathsf{R} LIN) \mid UL \\ UL &::= \mathsf{A}(LIN \mathsf{U} PSF) \mid \mathsf{A}(PSF \mathsf{U} UL) \mid UL \vee PSF) \mid PSF \vee UL) \\ PSF &::= PSF \wedge PSF \mid PSF \vee PSF \mid \neg(PSF) \mid * \end{aligned}$$

where PSF denotes the set of pure propositional formulas, and the R ("*release*") operator is the dual of the U operator, i.e., $s \models \varphi \mathsf{R} \rho \Leftrightarrow s \not\models \neg\varphi \mathsf{U} \neg\rho$ (see Section 4.6 for more discussion). The unique maximal set of ECTL templates whose instantiations guarantee linear witnesses are given by $\widetilde{LIN}$ [96], as the negation of LIN:

$$\begin{aligned} \widetilde{LIN} &::= PSF \mid \widetilde{LIN} \vee \widetilde{LIN} \mid \widetilde{LIN} \wedge PSF \mid PSF \wedge \widetilde{LIN} \mid \mathsf{EX}(\widetilde{LIN}) \mid \mathsf{E}(PSF \mathsf{U} \widetilde{LIN}) \mid \widetilde{UL} \\ \widetilde{UL} &::= \mathsf{E}(\widetilde{LIN} \mathsf{R} PSF) \mid \mathsf{E}(PSF \mathsf{R} \widetilde{UL}) \mid \widetilde{UL} \wedge PSF \mid PSF \wedge \widetilde{UL} \end{aligned}$$

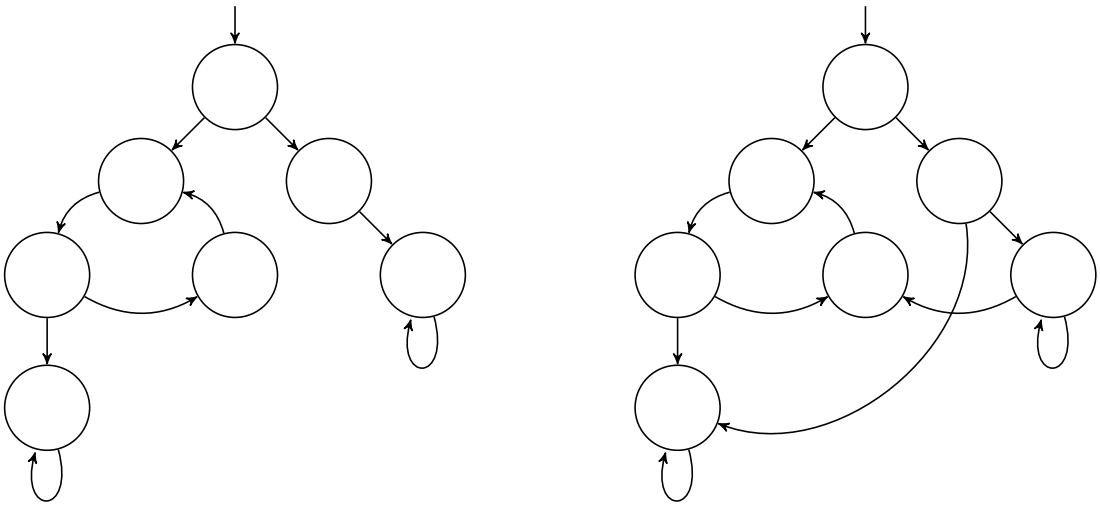
At the cost of restricting expressive power, ACTL and ECTL templates provide us a flexible syntax that makes it possible to formulate many ACTL specifications with linear counterexamples and ECTL specifications with linear witnesses in the usual manner. Optimization in model checking is possible by having the linearity guarantee [96].

2.2.1.2 Tree-like Counterexamples and Witnesses

The most significant disadvantage of linear counterexamples and witnesses is that we restrict the expressive power of branching time logic to its linear fragment. Clark et al. [27] suggested a

notion of tree-like counterexamples for ACTL. Intuitively, tree-like counterexamples are obtained by gluing together paths in a finite tree.

Let G be a directed graph. The *component graph* of G is a graph whose vertices are given by the strongly connected components (SCCs) in G, and where two vertices are connected by an edge if there exists an edge between vertices in the corresponding SCCs. A graph is *tree-like*, if all SCCs are either cycles or simple nodes, and the component graph is a directed tree (see Figure 2.8). A Kripke structure $(\mathcal{S}, \{s_{init}\}, \mathcal{N}, \mathcal{A}, \mathcal{L})$ is tree-like if the graph $(\mathcal{S}, \mathcal{N})$ is a finite tree-like graph whose root is the initial state s_{init}.



(a) (b)

Figure 2.8: (a) is tree-like, (b) is not.

Tree-like counterexamples are complete for ACTL, and tree-like witnesses are complete for ECTL. In other words, every violation of an ACTL formula or satisfaction of an ECTL formula can be described by tree-like structures. Consider an ECTL formula $\mathsf{EG}a \wedge \mathsf{EF}b$. A tree-like witness typically has the form illustrated in Figure 2.9. The left branch $s_0, s_1, s_2, s_3, ...$ demonstrates $\mathsf{EG}a$ and the right branch s_0, s_4, s_5 demonstrates $\mathsf{EF}b$.

Note that the concept of "tree-like" is used to describe models, or Kripke structures. It is well known that if a CTL formula has a model, this model is finite and its underlying graph is an infinite tree. The *tree-like model property* is stronger, requiring that the finite model is tree-like.

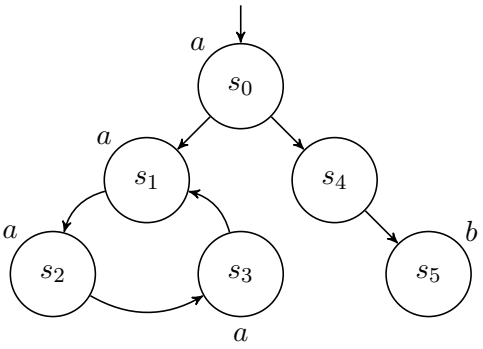


Figure 2.9: A tree-like witness for $\mathsf{EG}a \wedge \mathsf{EF}b$.

CHAPTER 3. GENERATION OF MINIMUM TREE-LIKE WITNESSES FOR EXISTENTIAL CTL

3.1 Introduction

Given a model checking problem, instead of merely answering "yes" or "no", model checkers may be able to return a witness or counterexample to verify satisfaction or violation of the specification, which is an important feature convincing engineers the significance of model checking. Since witnesses and counterexamples provide important debugging information and may be inspected by engineers, smaller ones are always preferable to ease their interpretation and understanding. Although much work has been published on witness or counterexample generation for CTL [32, 89, 58], to the best of our knowledge, no existing method guarantees their minimality for a general CTL formula with nested CTL operators. The use of backward exploration to verify EX, EF, and EU formulas inherently guarantees minimality of their linear witnesses, while a minimum lasso-shaped EG witness can be generated by computing transitive closures, for example using the saturation algorithm [103]. However, these approaches can only generate minimum linear witnesses for non-nested ECTL formulas and do not extend to general tree-like witnesses, which are complete for general ECTL formulas, as stated in Section 2.2.1.2. In other words, local minimality does not imply global minimality.

By recursively computing local fixpoints, the *saturation* algorithm [21] has shown clear advantages over traditional symbolic breadth-first approaches for state-space generation. It has also been applied to the computation of minimum EF [22] and EG [103] witnesses. In this chapter, we extend these ideas into a global approach to build minimum tree-like witnesses for arbitrary ECTL formulas.

This chapter is organized as follows. Section 3.2 summarizes background on decision diagrams, symbolic CTL model checking with decision diagrams, and the saturation algorithm. Section 3.3

defines the witness size and formalizes the computation of its minimum. Section 3.4 proposes saturation-based algorithms to symbolically encode minimum witness sizes for existential CTL operators, needed to obtain an overall minimum witness size. Section 3.5 describes how to generate a witness from the computed minimum witness size functions. Section 3.6 presents experimental results, and Section 3.7 concludes and outlines future work.

3.2 Background

3.2.1 Decision Diagrams

(Ordered) *binary decision diagrams* (BDDs) [12] are a compact data structure to represent and manipulate boolean formulas. Consider a domain $\mathcal{D}$ consisting of L boolean variables $\{v_1, v_2, ..., v_L\}$. An L-level BDD over $\mathcal{D}$ is a directed acyclic graph where:

- The only *terminal* nodes are the elements of $\mathbb{B} = \{\mathbf{0}, \mathbf{1}\}$ at level 0;
- A *nonterminal* node p at some level $p.lvl = i \in \{1, ..., L\}$ is associated with the domain variable $p.var = v_i$, and has two outgoing edges labeled 0 and 1, pointing to *children* $p[0]$ and $p[1]$ such that $p[0].lvl < i$ and $p[1].lvl < i$.

The BDD node p at level i encodes function $f_p : \mathbb{B}^L \to \mathbb{B}$, recursively defined as:

$$f_p(v_1, ..., v_L) = \begin{cases} f_{p[v_i]}(v_1, ..., v_L) & i \geq 1 \\ p & i = 0 \end{cases}.$$

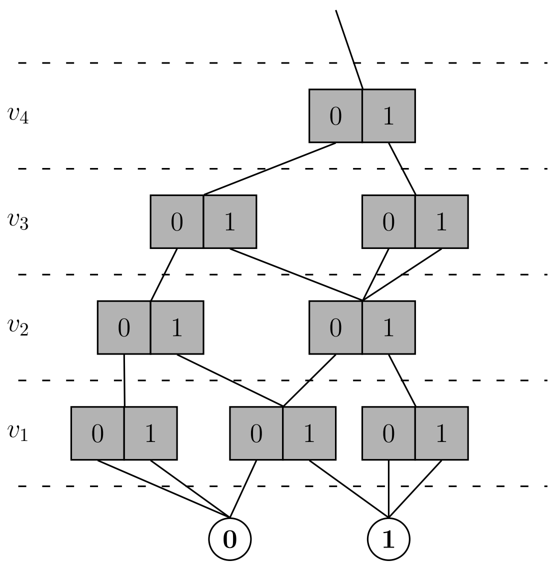
A BDD node p is said to be *redundant* if $p[0] = p[1]$. Two nodes p and q are said to be *duplicate* if $p.var = q.var$, $p[0] = q[0]$, and $p[1] = q[1]$. To ensure that each function has a unique BDD node representing it, we restrict ourselves to *canonical* forms of BDDs. Canonicity can be achieved by requiring that BDDs are either:

- *Quasi-reduced*: BDDs contain no duplicate nodes and do not skip variables. Let $p.var = v_i$. It is always true that $p[0].var = p[1].var = v_{i-1}$. If p is a root node, we have $i = L$.

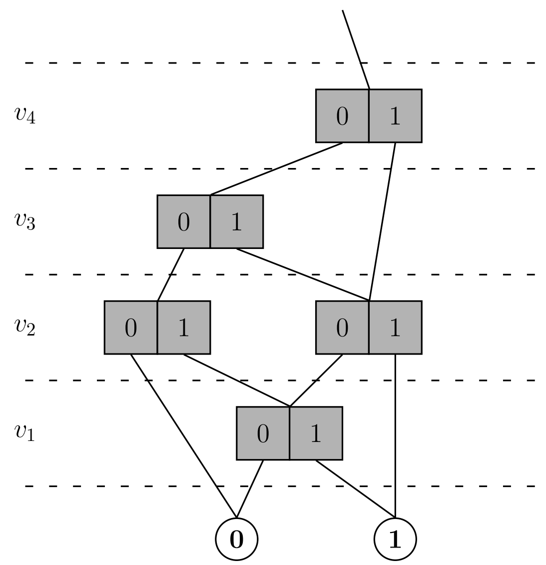
or

- *Fully-reduced*: BDDs contain no duplicate nodes and redundant nodes.

Other reduction rules such as zero-suppressed [68, 69] are not covered here. Figure 3.1 illustrates BDDs encoding the same boolean function in either quasi-reduced or fully-reduced form.



(a) A quasi-reduced BDD



(b) A fully-reduced BDD

Figure 3.1: BDDs in canonical forms.

Thanks to canonicity, checking equivalence of two boolean formulas is reduced to checking isomorphism between the corresponding two BDDs. A non-canonical BDD can be transformed into the *reduced* form but, in practice, one can avoid creating redundant and duplicate nodes during BDD manipulation. Identifying redundant nodes is simple. A *unique table*, usually a hash table [14], is employed to detect duplicates. If a newly created node p duplicates a node q stored in the unique table, q will be returned to replace p and p will be discarded. Otherwise, p is inserted into the unique table. Such node creation mechanism guarantees different nodes encodes different boolean functions, making equivalence checking possible in constant time.

On a more abstract level, a BDD node p can also encode a set $\mathcal{X}_p \subseteq \mathbb{B}^L$ of states through its *characteristic function*, i.e., $f_p(v_1, ..., v_L) = 1 \Leftrightarrow (v_1, ..., v_L) \in \mathcal{X}_p$. Therefore, the set of states

encoded by the BDDs in Figure 3.1 is:

$$\{(1,1,0,0),(1,0,1,0),(1,0,0,1),(1,0,1,1),(0,1,1,0),$$
$$(0,1,0,1),(0,1,1,1),(1,1,1,0),(1,1,0,1),(1,1,1,1)\}\,.$$

To encode relations over $\mathbb{B}^L$, we use $2L$-level BDDs over $(\mathbb{B}\times\mathbb{B})^L$, where the first boolean set in each pair corresponds to a "*from*", or "*unprimed*", local state and the second boolean set corresponds to a "*to*", or "*primed*", local state.

The size of a BDD can depend critically on the variable order. One example is the n-bit-comparator function for two integers $a_1a_2...a_n$ and $b_1b_2...b_n$, where a_i and b_i are the i-th bit of the corresponding integer [33]. If we choose the order $(a_1, b_1, ..., a_n, b_n)$, the resulting BDD has $3n + 2$ nodes. But if we choose the order $(a_1, ..., a_n, b_1, ..., b_n)$, the resulting BDD has $3 \cdot 2^n - 1$ nodes. In general, finding an optimal variable order is NP-hard [10]. Moreover, there are boolean functions that have exponential size BDDs for any variable ordering [13, 14]. Significant efforts have been made towards good ordering heuristics: static heuristics [40, 65, 1, 20, 85] establish a variable order using information available prior to building any BDDs, while dynamic heuristics [79, 9] attempt to improve the variable order by modifying the existing BDDs on the fly. There are occasions that require transforming existing BDDs to a different variable order. Research has also been done on such variable reordering problem through either rebuilding [90, 4, 80] or swapping adjacent variables [51].

We next explain how to manipulate BDDs. The key idea for efficient implementations of logical operations on BDDs is the *Shannon expansion*:

$$f = (\neg x \wedge f_{x\leftarrow 0}) \vee (x \wedge f_{x\leftarrow 1})\,,$$

where $f_{x\leftarrow 0}$ and $f_{x\leftarrow 1}$ are f with the argument x set equal to 0 and 1 respectively.

A uniform algorithm Apply() is given in Figure 3.2 for computing binary logical operations. For ease of exposition, we assume that BDDs are quasi-reduced. The two arguments, BDD nodes p and q, are the root nodes of the BDDs for boolean functions f and g. Let $\otimes$ be an arbitrary binary logical operator. The procedure *Apply* returns the root node u of the BDD for $f \otimes g$.

```
APPLY(BDD p, BDD q)
1: if both p and q are terminal nodes then return p ⊗ q          ▷ f and g are constant 0 or 1
2: if GET(p, q, u) then return u                                  ▷ retrieve previously computed result
3: k ← p.lvl                                                      ▷ we assume quasi-reduced rule, thus p.lvl = q.lvl ≥ 1
4: u ← BDDNODE(k)                                                 ▷ create a new BDD node at level k
5: u[0] ← APPLY(p[0], q[0])                                       ▷ compute f_{x←0} ⊗ g_{x←0}
6: u[1] ← APPLY(p[1], q[1])                                       ▷ compute f_{x←1} ⊗ g_{x←1}
7: u ← NORMALIZE(u)                                               ▷ ensure no duplicate nodes
8: PUT(p, q, u)                                                   ▷ memoize the result
9: return u
```

Figure 3.2: A uniform algorithm for computing binary logical operations on BDDs.

If f and g are constant boolean values, $f \otimes g$ is returned immediately (Line 1). Otherwise, we use the Shannon expansion:

$$f \otimes g = (\neg x \wedge (f_{x \leftarrow 0} \otimes g_{x \leftarrow 0})) \vee (x \wedge (f_{x \leftarrow 1} \otimes g_{x \leftarrow 1}))$$

to break the problem into two subproblems and solve each subproblem recursively (Line 5 and 6). If u duplicates a node that has already been included in the unique table, the procedure NORMALIZE() (Line 7) returns that node. Otherwise, it stores u in the unique table and returns u.

Since each subproblem can be broken into two subproblems, in order to prevent the algorithm from being exponential, APPLY() exploits dynamic programming: a *compute table* (or cache) [12] records previously computed subproblems (Line 8). Before any recursive call, the compute table is checked to see if the subproblem has been solved (Line 2). If it has, the result of the subproblem is retrieved from the compute table and returned; otherwise, the recursive call is performed. Thus, the number of subprobblems is bounded by the product of the size of the BDDs encoding f and g.

When the domain variable v_i has finite but not necessary boolean domain $\mathcal{S}_i = \{0, 1, ..., n_k - 1\}$ for some $n_k > 1$, we can use either *binary encoding* with $\lceil log_2 n_k \rceil$ boolean variables or *one-hot encoding* with n_k boolean variable (exactly one of which is set to 1) to represent v_i. An alternative is to use a natural extension of BDDs, (ordered) *multiway decision diagrams* (MDDs) [57]. The definition of an MDD is exactly as that of a BDD, except that the number of children of a node depends on its associated variable:

- For each $k \in \mathcal{S}_i$, a nonterminal node p associated with v_i has one outgoing edge labeled with k, pointing to a child $p[k]$.

In a MDD, a node p duplicates a node q if $p.var = q.var = v_i$, and $p[k] = q[k]$ for each $k \in \mathcal{S}_i$. The quasi-reduced and fully-reduced canonical forms are exactly analogous to those for BDDs. The MDD node p encodes function $f_p : \mathcal{S}_1 \times \mathcal{S}_2 \times ... \times \mathcal{S}_L \to \mathbb{B}$, recursively defined in a similar way as that for BDDs. We use $2L$-level MDD over $(\mathcal{S}_1 \times \mathcal{S}_1) \times (\mathcal{S}_2 \times \mathcal{S}_2) \times ... \times (\mathcal{S}_L \times \mathcal{S}_L)$, also called MDD2, to encode the corresponding transition relations.

Another extension of BDDs is (ordered) *additive edge-valued MDDs* (EV$^+$MDDs) [22], which can be used to encode partial integer-valued functions. An EV$^+$MDD is a directed acyclic edge-labeled and edge-valued graph where:

- The only terminal node is $\mathbf{\Omega}$ at level 0;

- A nonterminal node p at some level $p.lvl = i \in \{1, ..., L\}$ is associated with the domain variable $p.var = v_i$, and, for each $k \in \mathcal{S}_i$, has an outgoing edge with label k, pointing to a child $p[k].c$ at a level $p[k].c.lvl < i$, and value $p[k].v \in \mathbb{N}_\infty$.

For brevity, we write $p[k] = \langle p[k].v, p[k].c \rangle$.

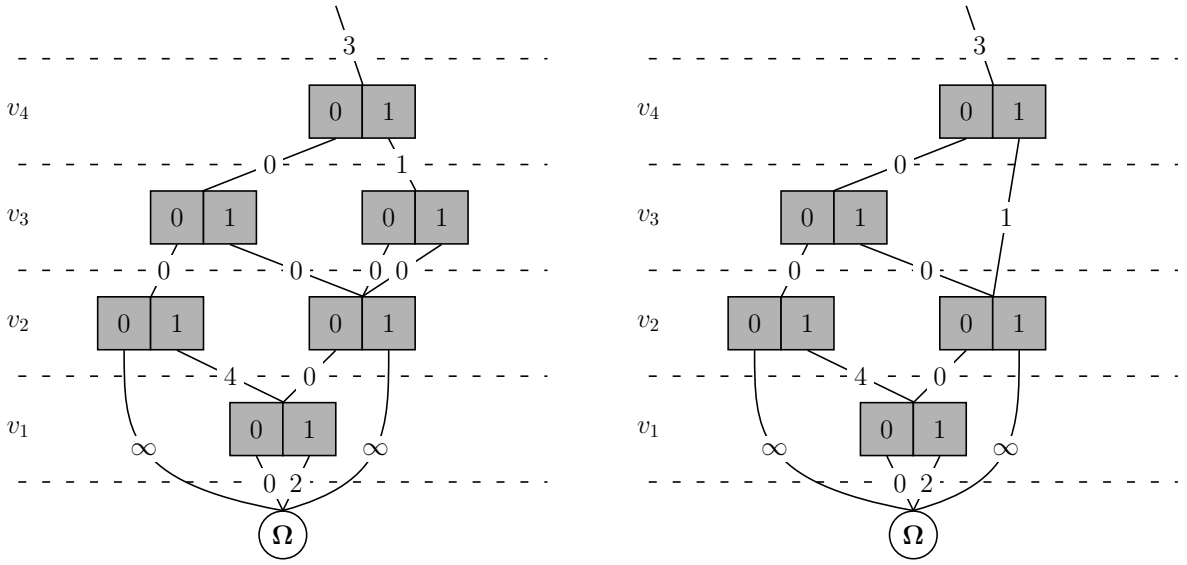
The EV$^+$MDD node p at level i encodes function $f_p : \mathcal{S}_1 \times \mathcal{S}_2 \times ... \times \mathcal{S}_L \to \mathbb{N}_\infty$, recursively defined as:

$$f_p(v_1, ..., v_L) = \begin{cases} p[k].v + f_{p[k].c}(v_1, ..., v_L) & i \geq 1 \\ 0 & i = 0 \end{cases},$$

where $k \in \mathcal{S}_i$. An EV$^+$MDD can also be viewed as associating an integer value, or infinity, to each state in $\mathcal{S}_1 \times \mathcal{S}_2 \times ... \times \mathcal{S}_L$. Similarly, $2L$-level EV$^+$MDDs, also called EV$^+$MDD2, encodes a partial integer-valued function over $(\mathcal{S}_1 \times \mathcal{S}_1) \times (\mathcal{S}_2 \times \mathcal{S}_2) \times ... \times (\mathcal{S}_L \times \mathcal{S}_L)$.

In order to achieve canonicity for EV$^+$MDDs, in addition, we require that each *normalized* nonterminal node must have at least one edge with value 0 and all edges with value ∞ must point to $\mathbf{\Omega}$. This means that the minimum value of the function encoded by any node is 0, but we can encode any partial function $f : \mathcal{S}_1 \times \mathcal{S}_2 \times ... \times \mathcal{S}_L \to \mathbb{N}_\infty$ with a "root edge" $\langle \sigma, p \rangle$, where σ is

the minimum value assumed by f, while the root node p encodes $f_p = f - \sigma$. See Figure 3.3 for EV$^+$MDDs encoding a function with minimum value 3 in either quasi-reduced or fully-reduced form.



(a) A quasi-reduced EV$^+$MDD

(b) A fully-reduced EV$^+$MDD

(v_1, v_2, v_3, v_4)	$f(v_1, v_2, v_3, v_4)$
0100	7
1100	9
0010	3
1010	5
0001	4
0011	4
1001	6
1011	6
...	∞

(c) The function being encoded

Figure 3.3: EV$^+$MDDs in canonical forms.

3.2.2 Symbolic CTL Model Checking with Decision Diagrams

In this section, we introduce symbolic CTL model checking, which uses decision diagrams to represent sets of states and transitions in Kripke structures and perform model checking [67]. It

manipulates sets rather than individual states and transitions and relies on the fixpoint characterization of the temporal logic operators.

We say that a set-valued function $f : \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S})$ is *monotonic* if, for all $\mathcal{X}, \mathcal{Y} \subseteq \mathcal{S}$, $\mathcal{X} \subseteq \mathcal{Y}$ implies $f(\mathcal{X}) \subseteq f(\mathcal{Y})$. A set $\mathcal{X} \subseteq \mathcal{S}$ is a *fixpoint* of f if $f(\mathcal{X}) = \mathcal{X}$. In particular, $\mathcal{X}$ is the *least fixpoint* of f if for any fixpoint $\mathcal{Y}$ of f, $\mathcal{X} \subseteq \mathcal{Y}$, and $\mathcal{X}$ is the *greatest fixpoint* of f if for any fixpoint $\mathcal{Y}$ of f, $\mathcal{X} \supseteq \mathcal{Y}$. A monotonic function f on $\mathcal{P}(\mathcal{S})$ always has a least and a greatest fixpoint [91], written as $\mu Z.f(Z)$ and $\nu Z.f(Z)$, respectively.

We write $f^i(Z)$ to denote i applications of f to Z. Formally,

$$f^i(Z) = \begin{cases} Z & i = 0 \\ f(f^{i-1}(Z)) & i \geq 1 \end{cases}.$$

The following lemma is the foundation of the algorithm to compute the least and the greatest fixpoints in model checking:

Lemma 3.2.0.1. *Provided that $\mathcal{S}$ is a finite set and f is monotonic on $\mathcal{S}$, the least fixpoint $\mu Z.f(Z) = \bigcup_i f^i(\emptyset)$ and the greatest fixpoint $\nu Z.f(Z) = \bigcap_i f^i(\mathcal{S})$.*

The corresponding algorithms are depicted in Figure 3.4. The computation of the least fixpoint starts with an empty set $\mathcal{X} = \emptyset$ and keeps augmenting $\mathcal{X}$ with $f(\mathcal{X})$ until $\mathcal{X}$ does not change, while the computation of the greatest fixpoint starts with a set containing all states $\mathcal{X} = \mathcal{S}$ and keep shrinking $\mathcal{X}$ with $f(\mathcal{X})$ until $\mathcal{X}$ does not change. $\mathcal{X}'$ stores the result from the previous iteration for comparison with the latest $\mathcal{X}$. Since $\mathcal{S}$ is a finite set, these algorithms are guaranteed to terminate within $|\mathcal{S}|$ iterations.

We identify the CTL formula φ with the set of states satisfying φ, i.e., $\{s \mid s \models \varphi\}$, in $\mathcal{P}(\mathcal{S})$. The basic CTL operators can be characterized as a least or greatest fixpoint of an appropriate

LFP(function f)
1: $\mathcal{X} \leftarrow \emptyset$
2: $\mathcal{X}' \leftarrow \mathcal{X}$
3: **repeat**
4: $\quad \mathcal{X}' \leftarrow \mathcal{X}$
5: $\quad \mathcal{X} \leftarrow f(\mathcal{X})$
6: **until** $\mathcal{X} = \mathcal{X}'$
7: **return** $\mathcal{X}$

(a) Computing the least fixpoint

GFP(function f)
1: $\mathcal{X} \leftarrow \mathcal{S}$
2: $\mathcal{X}' \leftarrow \mathcal{X}$
3: **repeat**
4: $\quad \mathcal{X}' \leftarrow \mathcal{X}$
5: $\quad \mathcal{X} \leftarrow f(\mathcal{X})$
6: **until** $\mathcal{X} = \mathcal{X}'$
7: **return** $\mathcal{X}$

(b) Computing the greatest fixpoint

Figure 3.4: Computing the least and the greatest fixpoints of a monotonic function f.

function [38]:

$$\mathsf{EF}\varphi = \mu Z.\varphi \vee \mathsf{EX}Z$$

$$\mathsf{E}(\varphi \mathsf{U} \rho) = \mu Z.\rho \vee (\varphi \wedge \mathsf{EX}Z)$$

$$\mathsf{EG}\varphi = \nu Z.\varphi \wedge \mathsf{EX}Z$$

$$\mathsf{AF}\varphi = \mu Z.\varphi \vee \mathsf{AX}Z$$

$$\mathsf{A}(\varphi \mathsf{U} \rho) = \mu Z.\rho \vee (\varphi \wedge \mathsf{AX}Z)$$

$$\mathsf{AG}\varphi = \nu Z.\varphi \wedge \mathsf{AX}Z$$

Intuitively, least fixpoints correspond to eventualities, while greatest fixpoints correspond to properties that should hold forever.

We elaborate the computation of sets of states satisfying EU and EG formulas. The computation for the remaining CTL operators can be established in a similar manner. Let $\mathcal{S}_\varphi$ be the set of states satisfying φ, i.e., $\{s \mid s \models \varphi\}$.

For EU formulas, since the application of EX is actually a previous state computation with $\mathcal{N}^{-1}$, the corresponding function f_{EU} is given by $f_{EU}(\mathcal{X}) = \mathcal{S}_\rho \bigcup (\mathcal{S}_\varphi \bigcap \mathcal{N}^{-1}(\mathcal{X}))$, with $\mathcal{X} = \emptyset$ initially, which returns the union of $\mathcal{S}_\rho$ and the intersection of $\mathcal{S}_\varphi$ and the set of states that can reach a state in $\mathcal{X}$ in one step. In other words, $f_{\mathsf{EU}}(\mathcal{X})$ is the set of states that either satisfy ρ or are, among predecessors of the states in $\mathcal{X}$, the ones satisfying φ. Consider the Kripke structure in Figure 3.5 and the formula $\mathsf{E}(a \mathsf{U} b)$. We initiate the computation with $\mathcal{X} = \emptyset$ and continuously

apply f_{EU} on the set of states obtained from the previous iteration, yielding an increasing sequence $f^0_{\mathsf{EU}}(\emptyset), f^1_{\mathsf{EU}}(\emptyset), f^2_{\mathsf{EU}}(\emptyset), ...$ such that $f^0_{\mathsf{EU}}(\emptyset) \subseteq f^1_{\mathsf{EU}}(\emptyset) \subseteq f^2_{\mathsf{EU}}(\emptyset) \subseteq ...$ due to the monotonicity of f_{EU}. This procedure terminates after four iterations, with the least fixpoint $\{s_0, s_2, s_3, s_4\}$:

$$
\begin{aligned}
&f^0_{EU}(\emptyset) = \emptyset \\
&f^1_{EU}(\emptyset) = f(\emptyset) && = \{s_2, s_4\} \\
&f^2_{EU}(\emptyset) = f(\{s_2, s_4\}) && = \{s_2, s_3, s_4\} \\
&f^3_{EU}(\emptyset) = f(\{s_2, s_3, s_4\}) && = \{s_0, s_2, s_3, s_4\} \\
&f^4_{EU}(\emptyset) = f(\{s_0, s_2, s_3, s_4\}) && = \{s_0, s_2, s_3, s_4\}
\end{aligned}
$$

Observing that the initial state $s_0 \models \mathsf{E}(a\mathsf{U}b)$, we claim that $\mathsf{E}(a\mathsf{U}b)$ is valid in this Kripke structure.

For EG formulas, the corresponding function f_{EG} for EG is given by $f_{\mathsf{EG}}(\mathcal{X}) = \mathcal{S}_\varphi \bigcap \mathcal{N}^{-1}(\mathcal{X})$, with $\mathcal{X} = \mathcal{S}$ initially, which returns the intersection of $\mathcal{S}_\varphi$ and the set of states that can reach a state in $\mathcal{X}$ in one step. In other words, $f_{\mathsf{EG}}(\mathcal{X})$ is the set of states that are, among predecessors of the states in $\mathcal{X}$, the ones satisfying φ. Consider the Kripke structure in Figure 3.5 and the formula $\mathsf{EG}a$. We initiate the computation with $\mathcal{X} = \mathcal{S} = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\}$ and continuously apply f_{EG} on the set of states obtained from the previous iteration, yielding a decreasing sequence $f^0_{\mathsf{EG}}(\mathcal{S}), f^1_{\mathsf{EG}}(\mathcal{S}), f^2_{\mathsf{EG}}(\mathcal{S}), ...$ such that $f^0_{\mathsf{EG}}(\mathcal{S}) \supseteq f^1_{\mathsf{EG}}(\mathcal{S}) \supseteq f^2_{\mathsf{EG}}(\mathcal{S}) \supseteq ...$ due to the monotonicity of f_{EG}. This procedure terminates after four iterations, with the greatest fixpoint $\{s_5, s_6\}$:

$$
\begin{aligned}
&f^0_{EG}(\mathcal{S}) = \mathcal{S} && = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\} \\
&f^1_{EG}(\mathcal{S}) = f(\mathcal{S}) && = \{s_0, s_3, s_5, s_6\} \\
&f^2_{EG}(\mathcal{S}) = f(\{s_0, s_3, s_5, s_6\}) && = \{s_0, s_5, s_6\} \\
&f^3_{EG}(\mathcal{S}) = f(\{s_0, s_5, s_6\}) && = \{s_5, s_6\} \\
&f^4_{EG}(\mathcal{S}) = f(\{s_5, s_6\}) && = \{s_5, s_6\}
\end{aligned}
$$

Observing that the initial state $s_0 \not\models \mathsf{EG}a$, we claim that $\mathsf{EG}a$ is not valid in this Kripke structure.

From the above, we can see that these algorithms are based on *backward* state traversal. The basic operation is to apply the previous-state function $\mathcal{N}^{-1}$ to get a set of predecessors. This is

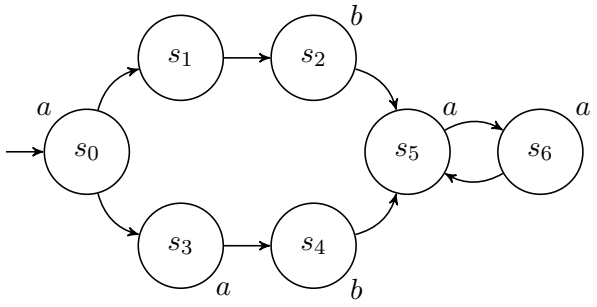


Figure 3.5: A Kripke structure M such that $M \models \mathsf{E}(a\mathsf{U}b)$ and $M \not\models \mathsf{EG}a$.

because we usually employ formalisms with future-time modalities, which are naturally evaluated by iterative application of $\mathcal{N}^{-1}$. Empirical evidence shows that, for some problems, symbolic model checking can perform better if it is based on *forward* state traversal [46, 43]; in this case, the next-state function $\mathcal{N}$ is applied to compute successors. The advantage of forward state traversal is that it only explores those parts of state space that are reachable from the initial state. However, due to the nature of decision diagrams, there is no guarantees that decision diagrams for forward state traversal are smaller or more efficiently manipulated than the ones for backward state traversal, even if they encode smaller sets of states.

The symbolic model checking algorithm is implemented as a procedure Check() that takes the CTL formula to be checked as the argument and returns a decision diagram that represents exactly the set of states that satisfy the formula. The Kripke structure being checked on is implicitly provided. Figure 3.6 defines Check() recursively over the structure of the given CTL formula φ. If φ is an atomic proposition $a \in \mathcal{A}$, Check() returns a decision diagram that represents the set of states satisfying a. The cases where φ is of the form $\neg\varphi'$, $\varphi' \wedge \rho'$, or $\varphi' \vee \rho'$ are handled using the standard algorithms for computing boolean connectives with decision diagrams. The subprocedure CheckEX() returns a decision diagram that represents the predecessors of the states satisfying φ' according to the transition relation, while CheckEU() and CheckEG() are implemented based on the least and the greatest fixpoint characterization. Since the other CTL operators can be rewritten using the operators above, the definition of Check() covers all CTL formulas.

CHECK(CTLFormula φ)
1: **if** $\varphi \in \mathcal{A}$ **then return** BDD(φ)
2: **if** $\varphi = \neg\varphi'$ **then return** $\neg$CHECK(φ')
3: **if** $\varphi = \varphi' \wedge \rho'$ **then return** CHECK(φ') $\cap$ CHECK(ρ')
4: **if** $\varphi = \varphi' \vee \rho'$ **then return** CHECK(φ') $\cup$ CHECK(ρ')
5: **if** $\varphi = \mathsf{EX}\varphi'$ **then return** CHECKEX(CHECK(φ'))
6: **if** $\varphi = \mathsf{E}(\varphi'\mathsf{U}\rho')$ **then return** CHECKEU(CHECK(φ'), CHECK(ρ'))
7: **if** $\varphi = \mathsf{EG}\varphi'$ **then return** CHECKEG(CHECK(φ'))

Figure 3.6: Symbolic model checking algorithm with decision diagrams for CTL.

3.2.2.1 Witness Generation

Clarke et al. [32] proposed the first symbolic witness generation approach for CTL. Of course, we cannot exhibit witnesses for universal formulas, only counterexamples, thus the presence of both existential and universal (non-negated) operators in a CTL formula φ means that we can neither provide a witness (in case φ holds) nor a counterexample (in case φ does not hold). We restrict our discussion to the existential fragment of CTL, i.e., ECTL.

Suppose that the state s satisfies the ECTL formula φ and we want to generate a witness to demonstrate it. First, consider the cases when φ is an unnested ECTL formula. If φ is a propositional formula, s itself is sufficient since φ does not require any state transition. A witness for $\mathsf{EX}a$ can be generated by presenting s and one of its successors that satisfy a, and is by definition minimal since all witnesses have size two (measured in the number of states).

Due to the fixpoint characterization, the symbolic model checking algorithm for $\mathsf{E}(a\mathsf{U}b)$ yields as an intermediate result the sequence of sets of states, $f^0_{\mathsf{EU}}(\emptyset), f^1_{\mathsf{EU}}(\emptyset), f^2_{\mathsf{EU}}(\emptyset), ...$, where $f^i_{EU}(\emptyset)$ is actually the set of states that can reach a state satisfying b through states satisfying a in at most $i-1$ steps. We can simply identify the minimum i such that $s \in f^i_{EU}(\emptyset)$, and generate a finite path $[\![s_0, s_1, s_2, ..., s_{i-1}]\!]$, where $s_0 \equiv s$ and $s_j \in f^{i-j}_{EU}(\emptyset)$ for every $j \in \{1, ..., i-1\}$. This is accomplished by finding the set of successors of s_j, intersecting it with $f^{i-j-1}_{\mathsf{EU}}(\emptyset)$, and then choose an arbitrary state from the resulting set as s_{j+1}. It is guaranteed that either $s_0 \models b$ (when $i = 1$),

or $s_0, s_2, s_3, ..., s_{i-1} \models a$ and $s_i \models b$ (when $i \geq 2$). Thus this finite path is a witness for $\mathsf{E}(a\mathsf{U}b)$, and is minimum due to the nature of breadth-first search.

A witness for $\mathsf{EG}a$ is lasso-shaped, consisting of a cycle and a finite path from s to that cycle, such that all states along that path and on the cycle satisfy a (See Figure 3.7) [7]. In other words, a state satisfying $\mathsf{EG}a$ must have a successor also satisfying $\mathsf{EG}a$; thus, we can incrementally build a path of states satisfying $\mathsf{EG}a$, which must finally lead to a state already on the path, closing the cycle and resulting in a witness. A witness generation algorithm for EG was proposed in [32] based on this idea. Since a state might have multiple successors satisfying $\mathsf{EG}a$, the algorithm is nondeterministic and the size of the witness depends on the state chosen at each step. While the algorithm uses a symbolic encoding, the approach is largely explicit, as it follows a single specific path. Decision diagrams help by efficiently encoding all states satisfying EG, but offer no help at all when deciding which of the states in $\mathcal{N}(s_i)$ satisfying EG should be chosen next, to continue the path from s_i.

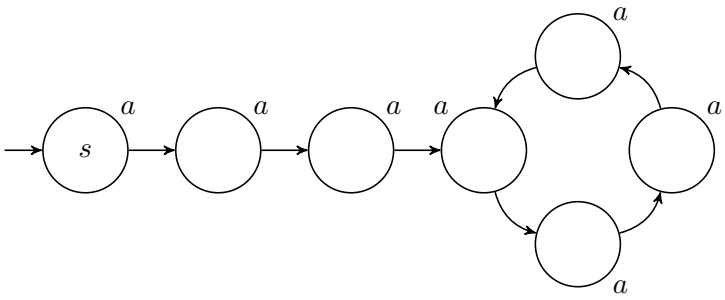


Figure 3.7: A witness for $\mathsf{EG}a$ is lasso-shaped.

Witness generation for general ECTL formulas, where existential CTL operators can be arbitrarily nested, is harder because the witness may not be linear but tree-like. Clark et al. [27] proposed algorithms to generate tree-like counterexamples for general (nested) ACTL formulas, or tree-like witnesses for general (nested) ECTL formulas. Since tree-like witnesses can be viewed as compositions of linear subwitnesses, they tie in well with the work for linear witnesses. As opposed to the bottom-up paradigm of symbolic model checking, the final witness is obtained by building and connecting subwitnesses for each subformula in a top-down process.

While computing the set of states satisfying $\mathsf{E}(\varphi\mathsf{U}\rho)$, again, the model checker retains the sequence of intermediate sets of states obtained from iterations of the least fixpoint computation. Recall that the intermediate set of states obtained from the i-th iteration contains the states that can reach a state satisfying ρ through states satisfying φ at most $i-1$ steps. Similar to what we have seen in the case of unnested EU formulas, when building a finite path witnessing $\mathsf{E}(\varphi\mathsf{U}\rho)$, these sets allow us to choose arbitrary successors satisfying $\mathsf{E}(\varphi\mathsf{U}\rho)$ but ensure the minimality of the fragment for $\mathsf{E}(\varphi\mathsf{U}\rho)$ in the final witness. Then, we build witnesses for ρ at the last state on the path and for φ at the other states, which can be tree-like, and glue them to the corresponding state to form a tree-like structure. The tree-like witness for $\mathsf{EG}\varphi$ is also built by gluing the path witnessing $\mathsf{EG}\varphi$ and a set of subwitnesses for φ.

The algorithm for EG is still nondeterministic and has no clue about the size of the final witness. Moreover, even when the ECTL formula does not contain EG and the final witness is linear, the minimality of the final witness is not guaranteed. An example can be found in Figure 3.8. Suppose that we are generating a witness for $\mathsf{EF}(\mathsf{EG}a)$. $\{s_2, s_3, s_4, s_5, s_6, s_7\}$ is the set of states satisfying $\mathsf{EG}a$. The greedy algorithm above searches for the shortest path to reach a state satisfying $\mathsf{EG}a$ and then a lasso-shaped witness for $\mathsf{EG}a$ from that state, yielding the composition of $[\![s_0, s_4]\!]$ and $[\![s_4, s_5, s_6, s_7, s_4]\!]$. However, a better choice is to glue $[\![s_0, s_1, s_2]\!]$ and $[\![s_2, s_3, s_3]\!]$ together because the result is smaller. Lacking of a global view of witness size, local minimality does not necessarily result in global minimality.

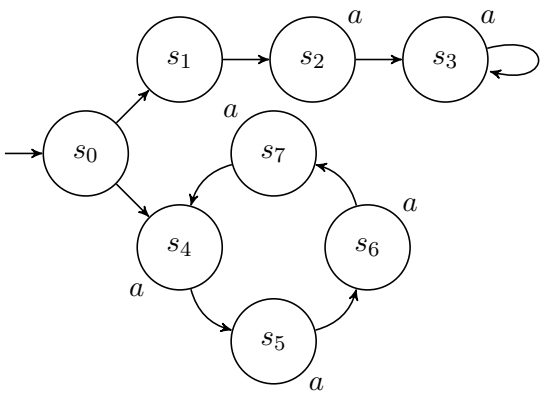


Figure 3.8: Local minimality does not necessarily result in global minimality.

3.2.3 The Saturation Algorithm

The traditional approaches of generating the reachable states of a system are based on a breadth-first traversal and apply a monolithic next-state function $\mathcal{N}$: after d iterations, the discovered state space contains states that are reachable from the initial state in at most d steps. This procedure terminates and obtains all the reachable states when the discovered state space does not change any more, as proved by the fixpoint theory.

It has been demonstrated that conjunctively or disjunctively partitioning the next-state function to be encoded by several decision diagrams can significantly improve the efficiency of symbolic model checking [16, 24]. For asynchronous systems, it is natural to store several next-state functions disjunctively according to a set $\mathcal{E}$ of asynchronous events, i.e., $\mathcal{N}_e$ for each $e \in \mathcal{E}$, so that the overall next-state function $\mathcal{N} = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e$ is never stored explicitly as a single decision diagram.

For an asynchronous system that is composed of multiple processes or components, we consider its state space $\mathcal{S}$ to be the product $\mathcal{S}_1 \times ... \times S_L$ of L finite state spaces, i.e., each *global* state $s \in \mathcal{S}$ is a tuple $(s[1], ..., s[L])$, where $s[k] \in \mathcal{S}_k$ is the *local* state for the k-th submodel. *Locality* is a fundamental property of this type of system: an event $e \in \mathcal{E}$ can be *independent* of the k-th submodel, i.e., its enabling condition does not depend on $s[k]$ and its firing condition does not change the value of $s[k]$.

Exploiting event locality, the saturation algorithm [21] has shown clear advantages over traditional symbolic breadth-first approaches for state space generation for asynchronous systems. The key idea is to compute local fixpoints using decision diagrams: fire events node-wise, bottom-up, and exhaustively, thereby bringing each node to a final *saturated* form. A saturated node encodes a fixpoint with respect to firing any event that is independent of all levels above that node's level, thus there is no need to visit it again when considering such events. The final reachable state space is always represented by a decision diagram that consists of saturated nodes with respect to all events only. For synchronous systems, an event depends on or affects every levels and thus the saturation algorithm regresses to a breadth-first approach.

Figure 3.9 presents the pseudocode to implement these steps. For simplicity, we assume quasi-reduction for MDDs. With saturation, we store a set of next-state functions $\{\mathcal{N}_1, ..., \mathcal{N}_L\}$ disjunctively to represent the overall next-state function $\mathcal{N} = \bigcup_{i=1}^{L} \mathcal{N}_i$, where $\mathcal{N}_i$ is independent of any variable above x_i. Given the input node p at level k, the procedure SATURATE() always returns a saturated node that encodes the fixpoint with respect to any event in $\{\mathcal{N}_1, ..., \mathcal{N}_k\}$. Line 5-6 saturates children of p to ensure that the children of the newly created node u are saturated. Thus firing any event in $\{\mathcal{N}_1, ..., \mathcal{N}_{k-1}\}$ at u does not yield any new state.. Line 7-11 fires $\mathcal{N}_k$ by invoking RELPRODSATURATE() and updating u repeatedly until reaching the fixpoint. The main difference between RELPRODSATURATE() and the normal procedure to compute the next states is Line 9, which guarantees the returned node to be saturated.

Though saturation is a heuristic approach and may not be optimal, it has many advantages:

- Firing $\mathcal{N}_k$ at u finds the maximum number of (partial) states under u, thanks to having the nodes below u saturated.
- Once a node at level k is saturated, we never fire an event in $\{\mathcal{N}_1, ..., \mathcal{N}_k\}$ at that node or any node reachable from it.
- The unique table and the compute table contain saturated nodes only, except the nodes in the MDD encoding $\mathcal{S}_{init}$.
- By definition, the final MDD can only contain saturated nodes. Saturation avoids unsaturated nodes that are created by the breadth-first approach but are never present in the final result.

Besides state space generation, the saturation algorithm has been employed for CTL model checking [23, 101], bounded model checking [97, 94, 34], computation of strongly connected components (SCC) [102], and minimum EF and EG witness generation [22, 103]. Meanwhile, variable ordering heuristics that improve the efficiency of saturation have also been proposed and investigated [84, 20, 85, 41].

SATURATE(MDD p)

1: $k \leftarrow p.lvl$
2: **if** $k = 0$ **then return** p ▷ p is a terminal node
3: **if** SATURATEGET(p, u) **then return** u ▷ SATURATE$(p) = u$
4: $u \leftarrow$ MDDNODE(k)
5: **for each** $i \in \mathcal{S}_k$ **do**
6: $\quad u[i] \leftarrow$ SATURATE$(p[i])$ ▷ ensure that children are saturated
7: **repeat**
8: $\quad$ **for each** $i, j \in \mathcal{S}_k$ **do**
9: $\quad\quad t \leftarrow$ RELPRODSAT$(u[i], \mathcal{N}_k[i][j])$
10: $\quad\quad u[j] \leftarrow$ UNION$(u[j], t)$
11: **until** u does not change
12: $u \leftarrow$ NORMALIZE(u)
13: SATURATEPUT(p, u) ▷ memoize the result
14: **return** u

RELPRODSATURATE(MDD p, MDD2 r)

1: $k \leftarrow p.lvl$
2: **if** $k = 0$ **then return** $p \wedge r$ ▷ p and r are terminal nodes
3: **if** RELPRODSATURATEGET(p, r, u) **then return** $\langle \alpha + \gamma, u \rangle$ ▷ RELPRODSATURATE$(p, r) = u$
4: $u \leftarrow$ MDDNODE(k)
5: **for each** $i, j \in \mathcal{S}_k$ **do**
6: $\quad t \leftarrow$ RELPRODSATURATE$(p[i], r[i][j])$
7: $\quad u[j] \leftarrow$ UNION$(u[j], t)$
8: $u \leftarrow$ NORMALIZE(u)
9: $u \leftarrow$ SATURATE(u)
10: RELPRODSATURATEPUT(p, r, u) ▷ memoize the result
11: **return** u

Figure 3.9: The saturation algorithm for state space generation.

3.3 Defining the Witness Size

We focus on the generation of witnesses for general (nested) ECTL formulas. As discussed in Section 2.2.1.2, these witnesses are finite tree-like Kripke structures and complete for ECTL. Recall that we only need to inspect their finite prefixes that are sufficient to explain the satisfaction. Here we give two definitions of *witness size*. The first one counts (distinct) states in the witness. The second one unfolds the witness to obtain its underlying graph, which is a finite computation tree where the same state may appear multiple times in different subwitnesses, and counts each appearance of any state.

For example, consider the (portion of a) Kripke structure shown in Figure 3.10(a). It is also a tree-like witness for $\mathsf{E}((\mathsf{EG}a)\mathsf{U}b)$, of size 5 according to the first definition. The underlying graph of the corresponding unfolded witness is shown in Figure 3.10(b), where the self-loop of state s_4 is repeated three times, once for each of states s_0, s_1, and s_2, since we need to show that each of them satisfies $\mathsf{EG}a$ (for clarity, a cycle is represented as a linear path along which the first and the last states are the same; dashed nodes represent the states closing cycles). Another way to think of this graph is that the first states of paths $[\![s_0, s_4, \overline{s_4}]\!]$, $[\![s_1, s_4, \overline{s_4}]\!]$, and $[\![s_2, s_4, \overline{s_4}]\!]$, each satisfying the inner formula $\varphi' = \mathsf{EG}a$, are "glued" onto the first three states of path $[\![s_0, s_1, s_2, s_3]\!]$, satisfying the outermost formula $\mathsf{E}(\varphi'\mathsf{U}\varphi'')$, as is the first (and only) state of path $[\![s_3]\!]$, satisfying the (atomic) inner formula $\varphi'' = b$. We write $\overline{s_4}$ for the last state of the EG witnesses to stress that this appearance of s_4 closes the cycle so that no more repeated appearances are needed. According to the second definition of witness size, since we counts the number of nodes in the resulting graph, a witness for a is path $[\![s_0]\!]$, of size 1, a witness for $\mathsf{EX}a$ is path $[\![s_0, s_1]\!]$, of size 2, a witness for $\mathsf{E}(a\mathsf{U}b)$ is path $[\![s_0, s_1, s_2, s_3]\!]$, of size 4, and a witness for $\mathsf{E}((\mathsf{EG}a)\mathsf{U}b)$ is the tree-like graph $[\![[\![s_0, s_4, \overline{s_4}]\!], [\![s_1, s_4, \overline{s_4}]\!], [\![s_2, s_4, \overline{s_4}]\!], s_3]\!]$, of size 10. For conjunction, we need additional path notation: a witness for $\mathsf{EX}a \wedge \mathsf{E}(a\mathsf{U}b)$ is a directed tree $[\![[\![s_0, s_4]\!] \Diamond [\![s_0, s_1, s_2, s_3]\!]]\!]$ with s_0 as the root, of size 5, where the separator $\Diamond$ indicates that the trees to its left and its right are to be merged on their root.

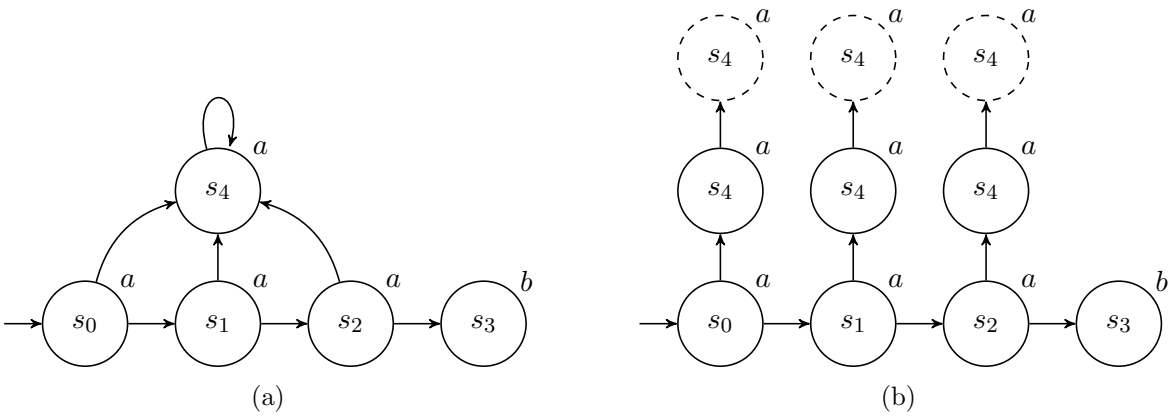


Figure 3.10: A Kripke structure satisfying $\mathsf{E}((\mathsf{EG}a)\mathsf{U}b)$ and its tree-like witness in two forms.

In this dissertation, we choose the second definition of witness size. To demonstrate the satisfaction of an nested ECTL formula, a state may appear multiple times for different purposes, which is more clear and easier to be presented in the unfolded witness. For example, the unfolded witness for $\mathsf{EF}(a \wedge \mathsf{EG}b)$ in Figure 3.11(b) contains state s_1 twice, once in $[\![s_2, s_1, \overline{s_2}]\!]$ to verify for the EG fragment, and once in $[\![s_0, s_1, s_2]\!]$ to verify the EF fragment. Considering each appearance separately in Figure 3.11(b) makes each subpath independently verifiable, while merging states that appear multiple times and counting only distinct states in Figure 3.11(a) loses this information. We believe that the appearance-based witness size is more appropriate when measuring the effort for inspecting a tree-like witness manually. Therefore, we assume the second definition that counts the appearances in the rest of this chapter.

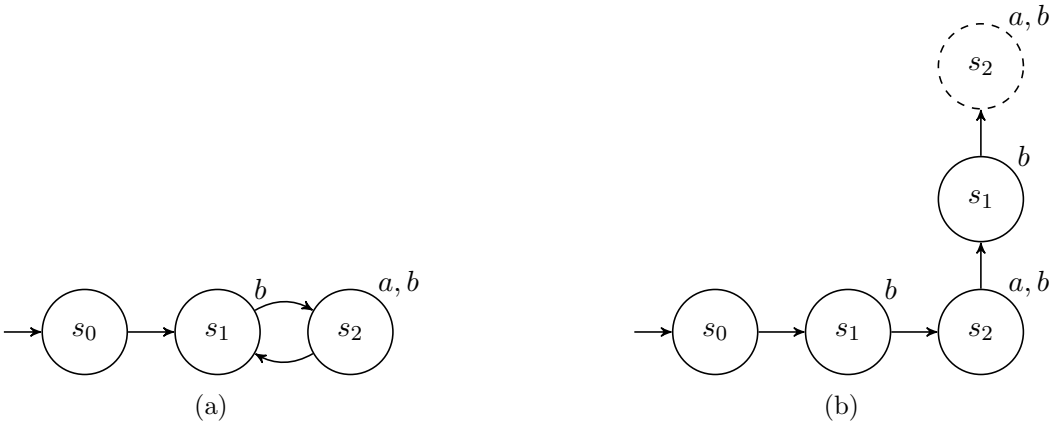


Figure 3.11: A Kripke structure satisfying $\mathsf{EF}(a \wedge \mathsf{EG}b)$ and its tree-like witness in two forms.

3.4 Computing the Minimum Witness Size

3.4.1 Minimum Witness Size Function

We recursively define function $\omega_\varphi : \mathcal{S} \to \mathbb{N}_\infty$ describing the minimum witness size for an ECTL formula φ starting from a state $s \in \mathcal{S}$ as follows:

$$\omega_a(s) = \begin{cases} 1 & \text{if } s \models a \\ \infty & \text{otherwise} \end{cases}$$

$$\omega_{\neg a}(s) = \begin{cases} \infty & \text{if } s \models a \\ 1 & \text{otherwise} \end{cases}$$

$$\omega_{\varphi \wedge \rho}(s) = \omega_\varphi(s) + \omega_\rho(s) - 1$$

$$\omega_{\varphi \vee \rho}(s) = min\{\omega_\varphi(s), \omega_\rho(s)\}$$

$$\omega_{\mathsf{EX}\varphi}(s) = min\{\omega_\varphi(s') : \forall s' \in \mathcal{N}(s)\} + 1$$

$$\omega_{\mathsf{E}(\varphi \mathsf{U} \rho)}(s) = min\{\omega_\rho(s), \omega_\varphi(s) + min\{\omega_{\mathsf{E}(\varphi \mathsf{U} \rho)}(s') : \forall s' \in \mathcal{N}(s)\}\}$$

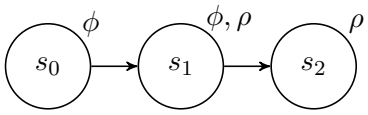
$$\omega_{\mathsf{EG}\varphi}(s) = min\{\chi_\varphi(s), \omega_\varphi(s) + min\{\omega_{\mathsf{EG}\varphi}(s') : \forall s' \in \mathcal{N}(s)\}\}$$

$$\chi_\varphi(s) = \begin{cases} min\{\sum_{i=1}^{n} \omega_\varphi(s_i) : \forall [\![s_0, s_1, ..., s_n \equiv s_0]\!] \in Cycle(s)\} + 1 & \text{if } Cycle(s) \neq \emptyset \\ \infty & \text{otherwise} \end{cases}$$

where $Cycle(s)$ is the set of cycles starting at s, and $\chi_\varphi(s)$ is the minimum witness size among cycles satisfying $\mathsf{EG}\varphi$ and starting from s.

Given a finite Kripke structure, if ECTL formula φ holds in the state s, we can always find a finite representation of witness that starts from s to demonstrate that. In the definition above, $\omega_\varphi(s) = \infty$, i.e., the representation of witness for $s \models \varphi$ is infinitely large, if and only if $s \not\models \varphi$. Meanwhile, the sum of ∞ and any integer or ∞ is equal to ∞. The logic behind this is that no witness exists if any subwitness does not exist.

If the given ECTL formula is an atomic proposition a or the negation of an atomic proposition $\neg a$ that holds in the state s, s itself is the witness, of size 1. For conjunction $\varphi \wedge \rho$, two subwitnesses are required to demonstrate the satisfaction of φ and ρ, respectively. We deduct 1 from the sum of



s	ω_φ	ω_ρ	$\omega^0_{\mathsf{E}(\varphi\mathsf{U}\rho)}$	$\omega^1_{\mathsf{E}(\varphi\mathsf{U}\rho)}$	$\omega^2_{\mathsf{E}(\varphi\mathsf{U}\rho)}$	$\omega^3_{\mathsf{E}(\varphi\mathsf{U}\rho)}$	$\omega^4_{\mathsf{E}(\varphi\mathsf{U}\rho)}$
s_0	5	∞	∞	∞	9	8	8
s_1	2	4	∞	4	3	3	3
s_2	∞	1	∞	1	1	1	1

Figure 3.12: An example to explain the computation of $\omega_{\mathsf{E}(\varphi\mathsf{U}\rho)}$

their sizes to avoid double counting the state s that they both start from. For disjunction $\varphi \vee \rho$, the smaller one of the two subwitnesses for φ and ρ suffices to serve as a witness. The witness for $\mathsf{EX}\varphi$ simply concatenates s and a witness for φ at one of s's successors.

We explain the computation of $\omega_{\mathsf{E}(\varphi\mathsf{U}\rho)}$ with an example in a breadth-first fashion. Consider the portion of a Kripke structure in Figure 3.12. Suppose that we already have the minimum witness size functions for the subformulas φ and ρ, which are presented in the columns ω_φ and ω_ρ in the table of Figure 3.12. Since $\omega_{\mathsf{E}(\varphi\mathsf{U}\rho)}$ is a least fixpoint, we use $\omega^i_{\mathsf{E}(\varphi\mathsf{U}\rho)}$ to denote the function obtained after the i-th iteration of the computation, and start from $\omega^0_{\mathsf{E}(\varphi\mathsf{U}\rho)}$, defined as $\omega^0_{\mathsf{E}(\varphi\mathsf{U}\rho)}(s) = \infty$ for any $s \in \mathcal{S}$. In the first iteration, the states satisfying ρ, i.e., s_1 and s_2, are identified to satisfy $\mathsf{E}(\varphi\mathsf{U}\rho)$), thereby $\omega^1_{\mathsf{E}(\varphi\mathsf{U}\rho)} = \omega_\rho$. In the second iteration, s_1's and s_2's predecessors satisfying φ, i.e., s_0 and s_1, are identified to satisfy $\mathsf{E}(\varphi\mathsf{U}\rho)$. s_0 has a witness for $\mathsf{E}(\varphi\mathsf{U}\rho)$, of size 9, by combining s_0's witness for φ, of size $\omega_\varphi(s_0) = 5$, and s_1's witness for $\mathsf{E}(\varphi\mathsf{U}\rho)$, of size $\omega^1_{\mathsf{E}(\varphi\mathsf{U}\rho)}(s_1) = 4$. Similarly, s_1 has a witness for $\mathsf{E}(\varphi\mathsf{U}\rho)$, of size 3, by combining s_1's witness for φ, of size $\omega_\varphi(s_1) = 2$, and s_2's witness for $\mathsf{E}(\varphi\mathsf{U}\rho)$, of size $\omega^1_{\mathsf{E}(\varphi\mathsf{U}\rho)}(s_2) = 1$. This new witness is smaller than the discovered one showing ρ at s_1 ($\omega^1_{\mathsf{E}(\varphi\mathsf{U}\rho)}(s_1) = 4$). Thus the value for s_1 in $\omega^2_{\mathsf{E}(\varphi\mathsf{U}\rho)}$ is reduced to 3, and the value for s_0, as s_1's predecessor satisfying ρ, is further reduced to 8 in the third iteration. There is no more change in the fourth iteration and the least fixpoint is reached.

A witness of $\mathsf{EG}\varphi$ is a lasso-shaped infinite path consisting of a finite prefix leading to a cycle [7]. Therefore, the computation of $\omega_{\mathsf{E}(\varphi\mathsf{U}\rho)}$ consists of two steps. First, among all states in the system, we identify the ones that exist in cycles of states satisfying φ, and their minimum witness size, which results in $\mathcal{X}_\varphi$. This step involves the computation of a transitive closure, to be discussed in Section 3.4.3. Second, from these states, we explore the system backward to find all states satisfying $\mathsf{EG}\varphi$, and their minimum witness size. This second step is essentially an EU computation.

We present symbolic algorithms to compute the minimum witness size for EU and EG, while the simpler ones for atomic propositions, logical operators, and EX are omitted. EV$^+$MDDs can encode partial integer functions and thereby are used to represent and manipulate minimum witness size functions symbolically.

3.4.2 Computing the Minimum Witness Size for EU Formulas

Though the example of computing $\omega_{\mathsf{E}(\varphi \mathsf{U} \rho)}$ is explained in a breadth-first fashion in Section 3.4.1, we adopt the saturation algorithm for its implementation, since saturation has shown clear advantages over traditional symbolic breadth-first approaches for fixpoint computation in asynchronous systems. In [101], a "constrained" variant of saturation that restricts exploration to states satisfying a given property was introduced. Instead of applying "after-the-fact" intersections, this approach employs a "check-and-fire" policy, firing an event only when the next states to be obtained satisfy the given property, through an on-the-fly check. Now, we further extend this idea to take into account the sizes of subwitnesses demonstrating the satisfaction of subformulas. Recall that the symbolic model checking algorithm for EU explores the system backwards. We use the previous-state function $\mathcal{N}^{-1}$, and assume that it is organized as a disjunction of a set of next-previous functions, $\mathcal{N}^{-1} = \bigcup_{i=1}^{L} \mathcal{N}_i^{-1}$, where $\mathcal{N}_i^{-1}$ is independent of any variable above x_i for every $i \in \{1, ..., L\}$.

EUSaturate() in Figure 3.13 is the top-level procedure to compute $\omega_{\mathsf{E}(\varphi \mathsf{U} \rho)}$, given $\langle \alpha_\rho, p_\rho \rangle$ encoding ω_ρ and $\langle \beta_\varphi, q_\varphi \rangle$ encoding ω_φ (both obtained by computing the minimum witness size function of subformulas). ConsSaturate() computes a fixpoint for the subfunction encoded by $\langle \alpha, p \rangle$, under constraint $\langle \beta, q \rangle$, with respect to events affected by or affecting variables up to p's level. The truth of the condition in Line 1 indicates that either no state can be explored backward from or the constraint is impossible to be met, and therefore we simply return the input subfunction $\langle \alpha, p \rangle$. ConsRelProdSaturate() first recursively computes the $\langle \beta, q \rangle$-constrained relational product of $\langle \alpha, p \rangle$ and r (specifically, it serves as a constrained version of the previous-state operation), and then saturates the resulting node to ensure that it encodes a local fixpoint.

EUSaturate(EV$^+$MDD $\langle \alpha_\rho, p_\rho \rangle$, EV$^+$MDD $\langle \beta_\varphi, q_\varphi \rangle$)

1: $\langle \mu, u \rangle \leftarrow$ ConsSaturate($\langle \alpha_\rho, p_\rho \rangle, \langle \beta_\varphi, q_\varphi \rangle$)
2: **return** $\langle \mu, u \rangle$

ConsSaturate(EV$^+$MDD $\langle \alpha, p \rangle$, EV$^+$MDD $\langle \beta, q \rangle$)

1: **if** $\alpha = \infty$ **or** $\beta = \infty$ **then return** $\langle \alpha, p \rangle$
2: $k \leftarrow p.lvl$ ▷ we assume the quasi-reduced rule, thus $p.lvl = q.lvl$
3: **if** $k = 0$ **then return** $\langle \alpha, \Omega \rangle$ ▷ $p = \mathbf{\Omega}$, $q = \mathbf{\Omega}$
4: **if** ConsSaturateGet($p, \langle \beta, q \rangle, \langle \gamma, u \rangle$) **then return** $\langle \alpha + \gamma, u \rangle$
5: $u \leftarrow$ EV$^+$MDDNode(k)
6: **for each** $i \in \mathcal{S}_k$ **do**
7: **if** $q[i].v = \infty$ **then** $u[i] \leftarrow p[i]$
8: **else** $u[i] \leftarrow$ ConsSaturate($p[i], \langle \beta + q[i].v, q[i].c \rangle$)
9: **repeat**
10: **for each** $i, j \in \mathcal{S}_k$ **do**
11: $\langle \tau, t \rangle \leftarrow$ ConsRelProdSaturate($\langle \alpha + u[i].v, u[i].c \rangle, \langle \beta + q[j].v, q[j].c \rangle, \mathcal{N}_k^{-1}[i][j]$)
12: $u[j] \leftarrow$ Min($u[j], \langle \tau, t \rangle$)
13: **until** u does not change
14: $\langle \mu, u \rangle \leftarrow$ Normalize(u)
15: ConsSaturatePut($p, \langle \beta, q \rangle, \langle \mu - \alpha, u \rangle$) ▷ memoize the result
16: **return** $\langle \mu, u \rangle$

ConsRelProdSaturate(EV$^+$MDD $\langle \alpha, p \rangle$, EV$^+$MDD $\langle \beta, q \rangle$, MDD2 r)

1: **if** $\alpha = \infty$ **or** $\beta = \infty$ **or** $r = \mathbf{0}$ **then return** $\langle \infty, \mathbf{\Omega} \rangle$
2: $k \leftarrow p.lvl$ ▷ we assume quasi-reduced rule, thus $p.lvl = q.lvl = r.lvl$
3: **if** $k = 0$ **then return** $\langle \alpha + \beta, \mathbf{\Omega} \rangle$ ▷ $r = \mathbf{1}$
4: **if** ConsRelProdSaturateGet($p, \langle \beta, q \rangle, r, \langle \gamma, u \rangle$) **then return** $\langle \alpha + \gamma, u \rangle$
5: $u \leftarrow$ EV$^+$MDDNode(k)
6: **for each** $i, j \in \mathcal{S}_k$ **do**
7: $\langle \tau, t \rangle \leftarrow$ ConsRelProdSaturate($\langle \alpha + p[i].v, p[i].c \rangle, \langle \beta + q[j].v, q[j].c \rangle, r[i][j]$)
8: $u[j] \leftarrow$ Min($u[j], \langle \tau, t \rangle$)
9: $\langle \mu, u \rangle \leftarrow$ Normalize(u)
10: $\langle \mu, u \rangle \leftarrow$ ConsSaturate($\langle \mu, u \rangle, \langle \beta, q \rangle$)
11: ConsRelProdSaturatePut($p, \langle \beta, q \rangle, r, \langle \mu - \alpha, u \rangle$) ▷ memoize the result
12: **return** $\langle \mu, u \rangle$

Figure 3.13: Algorithm to compute the minimum witness size for EU formulas.

When exploring the predecessors of the state s, we compute, for each predecessor $s' \in \mathcal{N}^{-1}(s)$, the sum of $\omega_\rho(s')$ and the value currently associated to s (Line 3 in ConsRelProdSaturate()), and use it to reduce the value associated to s', if smaller (Line 12 in ConsSaturate() and Line 8 in ConsRelProdSaturate()). Min() is the procedure to compute the element-wise minimum of two functions: for each $s \in \mathcal{S}$, $\text{Min}_{f,g}(s) = min\{f(s), g(s)\}$. Upon reaching the fixpoint at the top level, we have the size of the minimum tree-like $\mathsf{E}(\varphi \mathsf{U} \rho)$ witness for each state s (∞ if $s \not\models \mathsf{E}(\varphi \mathsf{U} \rho)$).

The hash-key for the cache entries of ConsSaturate() and ConsRelProdSaturate() consists of two nodes, p and q, plus the value β attached to the edge for q, representing the constraint. α just serves as an offset to all the values in the final function, while, for each individual state, it does not affect whether the new value obtained from one firing is smaller than the currently associated value. In other words, if it is known that $\text{ConsSaturate}(\langle 0, p\rangle, \langle \beta, q\rangle) = \langle \gamma, u\rangle$, then we can conclude that $\text{ConsSaturate}(\langle \alpha, p\rangle, \langle \beta, q\rangle) = \langle \alpha + \gamma, u\rangle$, for any $\alpha \in \mathbb{N}$. Excluding α from the hash-key tends to increase the "hits" of the corresponding entry in the compute table.

3.4.3 Computing the Minimum Witness Size for EG Formulas

As discussed before, the computation of $\omega_{\mathsf{EG}\varphi}$ consists of two steps. Since the second step is essentially an EU computation, we focus on the first step to compute $\mathcal{X}_\varphi$ in this section.

The *transitive closure* (TC) describes the reachability between any pair of nodes in a graph. Computing TCs was deemed infeasible for large models [78], but recent attempts using saturation to compute TCs symbolically have been successful [102, 103]. We generalize this approach so that, for a cycle of states satisfying φ, the size of a minimum witness for φ starting from each state on the cycle contributes to the cycle's overall size.

We compute function $TC_\varphi : \mathcal{S} \times \mathcal{S} \to \mathbb{N}_\infty$ such that $TC_\varphi(s, s')$ is the minimum overall size of any finite path $[\![s, s_1, ..., s']\!]$, computed as $\omega_\varphi(s_1) + ... + \omega_\varphi(s')$, or ∞ if no such path exists. Note that $\omega_\varphi(s)$ is not included because the goal of computing TC_φ is to obtain the minimum witness size of cycles, given by $\chi_\varphi(s) = TC_\varphi(s, s) + 1$, and $\omega_\varphi(s)$ should not be counted twice.

EGSaturate(EV$^+$MDD $\langle \beta_\varphi, q_\varphi \rangle$)

1: $\langle \tau, t \rangle \leftarrow$ TCSaturate(BuildLambda($\langle \beta_\varphi, q_\varphi \rangle$), $\langle \beta_\varphi, q_\varphi \rangle$)
2: $\langle \mu, u \rangle \leftarrow$ ConsSaturate(ExtractCycles($\langle \tau, t \rangle$), $\langle \beta_\varphi, q_\varphi \rangle$)
3: **return** $\langle \mu, u \rangle$

TCSaturate(EV$^+$MDD2 $\langle \alpha, p \rangle$, EV$^+$MDD $\langle \beta, q \rangle$)

1: **if** $\alpha = \infty$ **or** $\beta = \infty$ **then return** $\langle \alpha, p \rangle$
2: $k \leftarrow p.lvl$ ▷ we assume quasi-reduced rule, thus $p.lvl = q.lvl$
3: **if** $k = 0$ **then return** $\langle \alpha, \mathbf{\Omega} \rangle$ ▷ $p = \mathbf{\Omega}$, $q = \mathbf{\Omega}$
4: **if** TCSaturateGet($p, \langle \beta, q \rangle, \langle \gamma, u \rangle$) **then return** $\langle \alpha + \gamma, u \rangle$
5: $u \leftarrow$ EV$^+$MDD2Node(k)
6: **for each** $i, j \in \mathcal{S}_k$ **do**
7: $u[i].c[j] \leftarrow$ TCSaturate($\langle \alpha + p[i].v + p[i].c[j].v, p[i].c[j].c \rangle, \langle \beta + q[j].v, q[j].c \rangle$)
8: **for each** $i \in S_k$ **do**
9: $w \leftarrow$ EV$^+$MDD2Node($prime(k)$)
10: **repeat**
11: **for each** $j, j' \in \mathcal{S}_k$ **do**
12: $\langle \alpha', u' \rangle \leftarrow \langle \alpha + u[i].v + u[i].c[j].v, u[i].c[j].c \rangle$
13: $\langle \tau, t \rangle \leftarrow$ TCRelProdSaturate($\langle \alpha', u' \rangle, \langle \beta + q[j'].v, q[j'].c \rangle, \mathcal{N}_k[j][j']$)
14: $w[j'] \leftarrow$ TCMin($w[j'], \langle \tau, t \rangle$)
15: **until** w does not change
16: $u[i] \leftarrow$ Normalize(w)
17: $\langle \mu, u \rangle \leftarrow$ Normalize(u)
18: TCSaturatePut($p, \langle \beta, q \rangle, \langle \mu - \alpha, u \rangle$) ▷ memoize the result
19: **return** $\langle \mu, u \rangle$

Figure 3.14: Algorithm to compute the minimum witness size for EG formulas.

The procedure to compute TC_φ is analogous to a symbolic implementation of Dijkstra's algorithm in a weighted graph. We initialize the function

$$\lambda_\varphi(s_1, s_2) = \begin{cases} \omega_\varphi(s_2) & \text{if } s_2 \in \mathcal{N}(s_1),\ s_1 \models \varphi, \text{ and } s_2 \models \varphi \\ \infty & \text{otherwise} \end{cases}$$

and repeat the following computation until reaching a fixpoint:

$$\lambda_\varphi(s_1, s_2) = \min\{\lambda_\varphi(s_1, s_2), \min\{\lambda_\varphi(s_1, s) + \omega_\varphi(s_2) : \forall s \in \mathcal{N}^{-1}(s_2)\}\}.$$

This iteration never increases the value of λ_φ, thus it terminates, and when it does the resulting fixpoint is TC_φ. Procedures TCSaturate() and TCRelProdSaturate() in Figure 3.14

TCRelProdSaturate(EV$^+$MDD2 $\langle \alpha, p \rangle$, EV$^+$MDD $\langle \beta, q \rangle$, MDD2 r)

1: **if** $\alpha = \infty$ **or** $\beta = \infty$ **or** $r = 0$ **then return** $\langle \infty, \mathbf{\Omega} \rangle$
2: $k \leftarrow p.lvl$ ▷ we assume quasi-reduced rule, thus $p.lvl = q.lvl$
3: **if** $k = 0$ **then return** $\langle \alpha + \beta, \mathbf{\Omega} \rangle$ ▷ $p = \mathbf{\Omega}$, $q = \mathbf{\Omega}$
4: **if** TCRelProdSaturateGet$(p, \langle \beta, q \rangle, r, \langle \gamma, u \rangle)$ **then return** $\langle \alpha + \gamma, u \rangle$ ▷ $r = \mathbf{1}$
5: $u \leftarrow$ EV$^+$MDD2Node(k)
6: **for each** $i \in \mathcal{S}_k$ **do**
7: $w \leftarrow$ EV$^+$MDD2Node$(prime(k))$
8: **for each** $j, j' \in \mathcal{S}_k$ **do**
9: $\langle \alpha', p' \rangle \leftarrow \langle \alpha + p[i].v + p[i].c[j].v, p[i].c[j].c \rangle$
10: $\langle \tau, t \rangle \leftarrow$ TCRelProdSaturate$(\langle \alpha', p' \rangle, \langle \beta + q[j'].v, q[j'].c \rangle, r[j][j'])$
11: $w[j'] \leftarrow$ TCMin$(w[j'], \langle \tau, t \rangle)$
12: $u[i] \leftarrow$ Normalize(w)
13: $\langle \mu, u \rangle \leftarrow$ Normalize(u)
14: $\langle \mu, u \rangle \leftarrow$ TCSaturate$(\langle \mu, u \rangle, \langle \beta, q \rangle)$
15: TCRelProdSaturatePut$(p, \langle \beta, q \rangle, r, \langle \mu - \alpha, u \rangle)$ ▷ memoize the result
16: **return** $\langle \mu, u \rangle$

Figure 3.14: Algorithm to compute the minimum witness size for EG formulas (cont.).

are similar to ConsSaturate() and ConsRelProdSaturate() in Figure 3.13, except that they apply saturation to an EV$^+$MDD2. TCMin() (Line 14 in TCSaturate(), Line 11 in TCRelProdSaturate()) is an implementation of Min() over pairs of states: for each $s, s' \in \mathcal{S}$, $\text{TCMin}_{f,g}(s, s') = \min\{f(s, s'), g(s, s')\}$.

Finally, we compute $\omega_{\mathsf{EG}\varphi}$ with the procedure EGSaturate(), where $\langle \beta_\varphi, q_\varphi \rangle$ encodes ω_φ. BuildLambda() (Line 1) builds an EV$^+$MDD2 encoding function λ_φ, to initialize the computation of TC_φ. ExtractCycles() (Line 2) returns an EV+MDD encoding χ_φ by extracting elements $TC_\varphi(s, s)$ from TC_φ, for $s \in \mathcal{S}$, and adding 1 to them.

3.5 Generating a Minimum Tree-like Witness

Recall that, if function f is encoded as an EV$^+$MDD, one can retrieve the minimum value of f in constant time (as the value labeling the edge pointing to the root node) and a state achieving

that minimum value in time proportional to the number of levels L (by following a path of 0-valued edges from the root to $\mathbf{\Omega}$). Evaluating $f(s)$ for a given state s also requires just L steps.

Before generating a minimum overall witness, we determine if the model with initial state s_{init} satisfies the given ECTL formula φ_{E}. If $\omega_{\varphi_{\mathsf{E}}}(s_{init}) = \infty$, the model does not satisfy φ_{E} and thus no witness exists. Otherwise, there is a minimum witness of size $\omega_{\varphi_{\mathsf{E}}}(s_{init})$ for $s_{init} \models \varphi_{\mathsf{E}}$.

The pseudocode in Figure 3.15 describes how to generate one such minimum witness for $s \models \varphi$ recursively. Given a set of states $\mathcal{X}$ and an ECTL formula φ, GetStateWithMinWit() finds a state $s \in \mathcal{X}$ such that $\omega_\varphi(s) \leq \omega_\varphi(s')$ for any $s' \in \mathcal{X}$. This can be done by building an EV$^+$MDD encoding the function that returns 0 for every state in $\mathcal{X}$ or ∞ otherwise, computing the element-wise maximum of this function and ω_φ, and then choosing an arbitrary state with the minimum value from the resulting EV$^+$MDD, which encodes the function that returns $\omega_\varphi(s)$ if $s \in \mathcal{X}$ or ∞ otherwise. This procedure is invoked when we search for a successor that has the smallest minimum witness for the given ECTL formula among all successors of the current state. GetStateWithMinTC() is a similar procedure but it has an additional state t as argument, and finds a state $s \in \mathcal{X}$ such that $TC_\varphi(s,t) \leq TC_\varphi(s',t)$ for any $s' \in \mathcal{X}$. Since the witness for EG is lasso-shaped, we need the additional procedure CloseCycle() to find a cycle.

3.6 Experiments

We describe our experimental design in Section 3.6.1 and present the results in Section 3.6.2.

3.6.1 Experimental Design

We implemented both our EV$^+$MDD-based approach to generate minimum tree-like witnesses (MinWit) and the traditional MDD-aided breadth-first approach [27] to generate (not necessarily minimum) tree-like witnesses (Wit) in the model checker SMART [19]. Then we ran them on a benchmark suite consisting of nine Petri net models from the Model Checking Contest 2017 (https://mcc.lip6.fr/2017/). All models have one or more scaling parameters affecting the number of states and state-to-state transitions, thus the model size and complexity. To generate

GetMinWit(State s, ECTLFormula φ)

1: **if** $\varphi \in \mathcal{A}$ **then return** s ▷ φ is an atomic proposition
2: **if** $\varphi = \varphi' \wedge \rho'$ **then** ▷ $\omega_\varphi(s) = \omega_{\varphi'}(s) + \omega_{\varphi'}(s) - 1$
3: **return** $[\![\text{GetMinWit}(s, \varphi', \omega_{\varphi'}(s))]\!] \Diamond [\![\text{GetMinWit}(s, \rho', \omega_{\rho'}(s))]\!]$
4: **if** $\varphi = \varphi' \vee \rho'$ **then** ▷ $\omega_\varphi(s) = \min\{\omega_{\varphi'}(s), \omega_{\rho'}(s)\}$
5: **if** $\omega_{\varphi'}(s) < \omega_{\rho'}(s)$ **then return** $[\![\text{GetMinWit}(s, \varphi')]\!]$
6: **else return** $[\![\text{GetMinWit}(\mathbf{i}, \rho')]\!]$
7: **if** $\varphi = \mathsf{EX}\varphi'$ **then**
8: $s' = \text{GetStateWithMinWit}(\mathcal{N}(s), \varphi')$ ▷ there is at least one s' s.t. $\omega_{\varphi'}(s') = \omega_\varphi(s) - 1$
9: **return** $s, [\![\text{GetMinWit}(s', \varphi')]\!]$
10: **if** $\varphi = \mathsf{E}(\varphi' \mathsf{U} \rho')$ **then**
11: **if** $\omega_{\rho'}(s) = \omega_\varphi(s)$ **then** ▷ a minimum witness for $s \models \rho'$ works for $s \models \mathsf{E}(\varphi' \mathsf{U} \rho')$
12: **return** $[\![\text{GetMinWit}(s, \rho')]\!]$
13: **else** ▷ no witness for $s \models \rho'$ is minimum for $s \models \mathsf{E}(\varphi' \mathsf{U} \rho')$
14: $s' = \text{GetStateWithMinWit}(\mathcal{N}(s), \varphi)$ ▷ there is at least one s' s.t. $\omega_\varphi(s') = \omega_\varphi(s) - \omega_{\varphi'}(s)$
15: **return** $[\![\text{GetMinWit}(s, \varphi')]\!], \text{GetMinWit}(s', \varphi)$
16: **if** $\varphi = \mathsf{EG}\varphi'$ **then**
17: **if** $TC_{\varphi'}(s, s) = \omega_\varphi(s) - 1$ **then** ▷ a minimum cycle witness works for $s \models \mathsf{EG}\varphi'$
18: **return** $\text{CloseCycle}(s, s, \varphi')$
19: **else** ▷ s is on the handle of a lasso for a minimum witness for $s \models \mathsf{EG}\varphi'$
20: $s' = \text{GetStateWithMinWit}(\mathcal{N}(s), \varphi)$ ▷ there is at least one s' s.t. $\omega_\varphi(s') = \omega_\varphi(s) - \omega_{\varphi'}(s)$
21: **return** $[\![\text{GetMinWit}(s, \varphi')]\!], \text{GetMinWit}(s', \varphi)$

CloseCycle(State s, State t, ECTLFormula φ)

1: **if** $TC_\varphi(s, t) = \omega_\varphi(t)$ **then** ▷ it must be that $t \in \mathcal{N}(s)$, close the cycle with $\bar{t}$
2: **return** $[\![\text{GetMinWit}(s, \varphi)]\!], \bar{t}$
3: **else**
4: $s' = \text{GetStateWithMinTC}(\mathcal{N}(s), t, \varphi)$ ▷ there is at least one s' s.t. $\omega_\varphi(s') = TC_\varphi(s, t) - TC_\varphi(s', t)$
5: **return** $[\![\text{GetMinWit}(s, \varphi)]\!], \text{CloseCycle}(s', t, \varphi)$

Figure 3.15: Algorithm to generate a minimum tree-like witness.

tree-like witnesses, we define an ECTL formula that the model satisfies (the specific formula is listed in the results presented in Table 3.1). The datasets we utilized are available in the figshare repository [52].

For MinWit, we run each model instance with a timeout of one hour, and report the runtime, the peak memory consumption, and the size of the minimum witness. For Wit, we run each instance 100 times and report the total runtime, the peak memory consumption, and the minimum, average and maximum size among the all the generated witnesses. The minimum witness size is in bold when Wit did not manage to generate a minimum witness in any of the 100 runs. Obviously, the choice of $R = 100$ runs is arbitrary: the larger R is, the more likely Wit is to generate smaller witnesses, possibly a minimum one, but, on the other hand, the overall time Wit spends on witness generation is roughly proportional to R. Fundamentally, however, we have no easy way to know if the smallest witness generated by Wit is a minimum one, regardless of how large R is, while MinWit guarantees minimality.

3.6.2 Experimental Results

The experimental results are presented in Table 3.1. A few observations are in order. First, it is not surprising that MinWit is sometimes orders of magnitude slower and requires more memory than Wit. Building EV^+MDDs or EV^+MDD2s encoding both states and size information is much more expensive than the image computations on MDDs used to just run the model checking phase, as Wit does. However, this is offset by a minimality guarantee. Interestingly, there are cases where MinWit completes with a runtime and memory consumption comparable to a single run of Wit (e.g., `Kanban`) or even faster (e.g., `SmallOS`). We give credit to the saturation algorithm for its efficient locality-exploiting exploration.

Second, for models where small, simple witnesses exist, Wit may be able to generate a minimum witness. Since the backward exploration guarantees the local minimality of subwitnesses for EX, EF and EU segments, such greedy strategy may result in a global minimum witness, determined by the structure of the model. But this occurrence cannot be guaranteed, regardless of whether

Table 3.1: Performance comparison of MinWit and Wit.

Model(params)	#States	#Trans	Time (s)		Memory (MB)		Size			
			MinWit	Wit	MinWit	Wit	MinWit	Wit		
								min	avg	max
$\mathsf{EG}(\mathsf{EF}((Section\_2 = 1) \wedge (Section\_3 = 1)))$										
CircularTrain(12)	$2.0 \cdot 10^2$	$5.0 \cdot 10^2$	0.4	0.0	20.6	4.8	**25**	32	71	275
CircularTrain(24)	$8.7 \cdot 10^4$	$4.1 \cdot 10^5$	2244.9	4.6	2924.5	11.8	**37**	91	404	889
$\mathsf{E}(\mathsf{EF}(ERKPP > 5)\,\mathsf{U}\,\mathsf{EG}(RKIPP\_RP > 5))$										
ERK(20)	$1.7 \cdot 10^6$	$1.6 \cdot 10^7$	93.9	3.8	591.0	6.5	**129**	258	313	391
ERK(22)	$2.8 \cdot 10^6$	$2.7 \cdot 10^7$	224.1	4.4	932.1	6.9	**129**	246	314	393
ERK(25)	$5.7 \cdot 10^6$	$5.4 \cdot 10^7$	793.4	5.0	1800.6	8.0	**129**	256	315	397
$\mathsf{EF}((P1 = 3) \wedge \mathsf{EG}((P1 > P2) \wedge (P2 > P3)))$										
FMS(5)	$2.9 \cdot 10^6$	$3.2 \cdot 10^7$	0.6	2.7	21.0	7.6	13	13	48	193
FMS(8)	$2.5 \cdot 10^8$	$3.6 \cdot 10^9$	31.6	17.5	447.6	13.7	22	22	51	201
FMS(10)	$2.5 \cdot 10^9$	$4.1 \cdot 10^{10}$	458.9	30.6	1510.1	23.5	28	28	54	217
$\mathsf{EF}((P1 < P2) \wedge \mathsf{EG}(P1 = P4))$										
Kanban(20)	$8.1 \cdot 10^{11}$	$1.1 \cdot 10^{13}$	18.4	1530.3	269.6	289.0	10	10	10	11
Kanban(22)	$2.1 \cdot 10^{12}$	$2.9 \cdot 10^{13}$	28.6	2297.8	395.9	410.4	10	10	10	11
Kanban(25)	$7.7 \cdot 10^{12}$	$1.1 \cdot 10^{14}$	53.9	4224.4	691.6	707.9	10	10	10	11
$\mathsf{E}(\mathsf{EF}(Phase1 < Phase2)\,\mathsf{U}\,(Phase2 > Phase3))$										
MAPK(8)	$6.1 \cdot 10^6$	$7.9 \cdot 10^7$	14.0	2.0	353.0	6.1	**70**	126	126	126
MAPK(12)	$3.2 \cdot 10^8$	$5.0 \cdot 10^9$	1881.4	6.2	1764.2	8.7	**109**	204	204	204
$\mathsf{EF}((Think\_1 = 0) \wedge \mathsf{EG}(Eat\_1 = 0))$										
Philosophers(20)	$3.5 \cdot 10^9$	$5.4 \cdot 10^{10}$	1.3	2.1	52.4	9.2	5	5	8	22
Philosophers(50)	$7.2 \cdot 10^{23}$	$2.8 \cdot 10^{25}$	44.9	10.8	763.5	29.1	5	5	7	20
Philosophers(100)	$5.2 \cdot 10^{47}$	$4.0 \cdot 10^{49}$	timeout	52.1	-	94.0	-	5	7	28
$\mathsf{E}(\mathsf{EF}(P\_client\_ack\_1 = 1)\,\mathsf{U}\,((P\_server\_ack\_1 = 1) \wedge (P\_server\_ack\_2 = 1)))$										
SimpleLoadBal(2)	$8.3 \cdot 10^2$	$3.4 \cdot 10^3$	0.1	0.8	9.0	5.9	23	23	32	44
SimpleLoadBal(5)	$1.2 \cdot 10^5$	$7.5 \cdot 10^5$	37.6	19.2	1032.6	41.0	**23**	26	69	80
$\mathsf{E}(\mathsf{EF}(TaskOnDisk < CPUUnit)\,\mathsf{U}\,(CPUUnit < DiskControllerUnit))$										
SmallOS(64,32)	$9.1 \cdot 10^6$	$6.8 \cdot 10^7$	17.5	1987.7	374.6	401.6	**662**	694	1189	1552
SmallOS(128,64)	$2.6 \cdot 10^8$	$2.0 \cdot 10^9$	294.4	53522.2	3228.4	1850.0	**2342**	2430	4698	5920
$\mathsf{EF}(\mathsf{EG}(Undress < InBath))$										
SwimmingPool(1)	$9.0 \cdot 10^4$	$4.5 \cdot 10^5$	109.7	4.7	1334.9	6.6	16	16	24	43
SwimmingPool(2)	$3.4 \cdot 10^6$	$2.0 \cdot 10^7$	timeout	39.1	-	22.3	-	16	24	53

we use 100 runs or 10,000 runs, so that, even when Wit happens to generate a minimum witness, users do not know that this is indeed the case.

Third, for models where only large, complex witnesses exist, generating a minimum witness is almost impossible for Wit, while the witness from MinWit may be as small as 40% of the size of the smallest one generated by Wit (e.g., `CircularTrain` with $N = 24$). Additionally, Wit's greedy strategy may trap itself into a local optimum. For example, the ECTL formula used for model `MAPK` does not contain EG, and the minimum, average, and maximum witness sizes generated by Wit are equal, implying that Wit is unaware of other possibilities when it chooses branching states. Adopting a probabilistic non-optimal strategy like simulated annealing may alleviate this problem, but it still would not provide guarantees and would likely require many more runs.

The main limitation of MinWit is that, since computing the minimum witness size function is computationally intensive, long runtimes and large amounts of memory are required as the model complexity scales up. However, engineers usually debug models with small scaling parameters first, perhaps running model checking tools overnight, thus, the resource requirement of MinWit may often be acceptable in practice. In real-world applications, we believe that MinWit and Wit can complement each other. Wit generates a large number of witnesses in a short time, but if all the witnesses are complex, MinWit can be run to find a smaller, easier-to-inspect one. Conversely, if MinWit fails to generate a minimum witness due to time or memory limitation, Wit can be run to obtain not-necessarily-minimum ones by running it repeatedly, as many times as one can afford. The best approach, given enough resources and in the presence of critical deadlines, may well be to run both methods in parallel, so that we can be sure to have a minimum witness if MinWit completes, but we have at least some witnesses from Wit, if MinWit fails to complete in time.

3.7 Conclusions

We presented a definition of witness size and an approach to compute minimum tree-like witnesses for ECTL formulas, based on edge-valued decision diagrams to capture a global view of witness size. Experimental results demonstrate that our approach MinWit is able to generate

minimum witnesses (with a guarantee that it is doing so) for some models, while the traditional breadth-first approach Wit is not. While the runtime and memory requirements of MinWit tend to be higher, sometimes they are comparable to that of Wit.

There are several directions for future work to improve this approach itself or extend its applicability. One interesting possibility is to selectively employ MinWit or Wit for different subformulas; this would not guarantee witness minimality, but could generate smaller witness than with Wit alone, while being faster than using MinWit alone. Especially for EG formulas, Wit has no global view about the size of witnesses it generates, while, for formulas where the minimum witness size from each state varies widely, the EV$^+$MDDs and EV$^+$MDD2s built by MinWit tend to be large and costly to compute. Thus, heuristics that consider both the structure of the model and of the formula are needed.

Another possible direction is to exploit the observation that Wit can be orders of magnitude faster than MinWit. We may run Wit a few times and use the size of the smallest witness as an upper bound of the minimum size. Then in the iterative process of fixpoint computation in MinWit, for (sub-)states with a minimum (sub-)witness of size equal to or larger than the upper bound, it is not necessary to explore further from them and thus the search space is pruned. This idea may leverage the findings in [97]. Storing distance information in an EV$^+$MDD, their approach of bounded reachability checking performs forward state space exploration by first computing the next states, followed by an approximate but fast truncation to prune (sub-)states with at least one edge value exceeding the given bound B. As a result, the EV$^+$MDD may contain states with distance up to $B \times L$, where L denotes the number of levels. It turned out to outperform the exact truncation that prunes (sub-)states whose sum of the corresponding edge values exceeds B. Adopting similar approximate truncation may help prune the search space for MinWit efficiently, given an upper bound of minimum witness size obtained from Wit.

Finally, MinWit could be further extended by generalizing the concept of "size" to "weight". Specifically, by assigning a weight to each state, engineers could convey their preference to model checkers, which would then tend to generate witnesses containing the desired states and paths,

instead of just counting the number of states in the witness. The algorithmic difference in doing so would be negligible, the only additional cost could be a potential growth in the size of the corresponding EV^+MDDs and EV^+MDD2s, as the functions being encoded might have less sharing of nodes.

CHAPTER 4. BOUNDED MODEL CHECKING FOR EXISTENTIAL CTL

4.1 Introduction

SAT-based *bounded model checking* (BMC) is a complementary technique to decision-diagram-based model checking [17]. By exploiting the observation that many real-life models have "shallow" witnesses, BMC only considers finite prefixes of infinite paths. It translates the semantics of temporal logic, bounded by some integer k, into a propositional formula, and leverages a SAT solver to check for satisfiability. If the formula is determined to be satisfiable, a witness can be generated from the truth assignment produced by the SAT solver. Otherwise, k is increased progressively until either a witness is found, or some preset upper bound is reached. BMC solves the model checking problem by searching for witnesses and thus inherently has the capability to generate witnesses.

BMC was first proposed for linear temporal logic (LTL) [7, 8] and later applied to the universal fragment of computation tree logic (ACTL) [74] and $\forall\mu$-calculus [95, 71]. Decision-diagram-based techniques inspired by the same idea were also proposed [97, 94]. We concentrate on SAT-based BMC for ECTL, which is different from applying BMC to LTL, since the general form of ECTL witnesses is tree-like [27]. In [74, 98, 18], such a witness in a bounded model is represented as a set of bounded paths. Different encoding schemes based on proof systems were proposed in [95, 71], in the larger context of $\forall\mu$-calculus. In this chapter, we improve the translation to propositional formulas from [74, 98, 18] by reducing the number of bounded paths that must be considered. Our approach generates a smaller formula, which is often easier for a SAT solver to answer, or the same one as in the classic approach, in the worst case. In addition, we propose a simple modification to the translation so that it is also defined for models with deadlock states.

The remainder of this chapter is organized as follows. Section 4.2 introduces SAT solvers and bounded model checking, and explains bounded model checking for ECTL in detail. Section 4.3

proposes an improved translation to produce a possibly smaller propositional formula. Section 4.4 compares the classic approach and ours with respect to the minimum bound to find a witness, which determines the earliest possibility for BMC to provide an answer, and with respect to the complexity of propositional formulas, in terms of the number of symbolic states considered to form a witness. Section 4.5 modifies the translation to cope with models containing deadlock states. Section 4.7 describes our experimental design and presents the results, while Section 4.8 concludes the discussion and outlines future work.

4.2 Background

We present a brief introduction of SAT solvers in Section 4.2.1 and bounded model checking, specializing in the context of ECTL, in Section 4.2.2.

4.2.1 SAT Solvers

In this section, we first introduce the *propositional satisfiability* (SAT) problem and then the implementation of SAT solvers.

In propositional logic, the basic element is a *boolean variable*, or *propositional variable*, whose value can be either **true** or **false**. A *boolean formula*, or *propositional formula*, is a combination of boolean variables and connectives including a unary operator *negation* ($\neg$) and four binary operators *conjunction* ($\wedge$), *disjunction* ($\vee$), *implication* ($\Rightarrow$), and *equivalence* ($\Leftrightarrow$). Given a set of variables $\mathcal{X} = \{x_1, ..., x_n\}$, a *variable assignment* is a function: $\mathcal{X} \rightarrow \mathbb{B}$ that gives values to variables. The *propositional satisfability* problem (SAT) is to determine whether there exists a variable assignment that evaluates the given propositional formula to **true**.

As the first and the most well-known NP-complete problem, SAT is important with respect to both theoretical studies and practical applications, motivating the research and development of SAT solvers as specialized tools. Since their inception in the mid-1990s, modern SAT solvers (such MiniSat [37], Lingeling [6], Glucose [2], and CryptoMiniSat [86]) are able to solver SAT problems with millions of variables and clauses in an acceptable time and have been applied to a number of

areas, for example, AI planning [59], scheduling [99, 44], electronic design automation (EDA) [87], software verification [47], and model checking [7].

Modern SAT solvers accept propositional formulas in *conjunctive normal form* as the standard input. A *literal* is either a boolean variable x or its negation $\neg x$, and a *clause* is a disjunction of literals. A propositional formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses. For convenience, a CNF formula can also be viewed as a set of clauses, and a clauses as a set of literals. Every propositional formula can be transformed into CNF. The naive transformation approach is to use De Morgan's laws, $\neg(x \wedge y) = \neg x \vee \neg y$, $\neg(x \vee y) = \neg x \wedge \neg y$, and the distributive property, $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$, $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$). However, this can cause an exponential growth in the formula size. A better approach is Tseitin transformation [77, 93], which increases the formula size linearly. For each subformula, a new variable representing it is introduced and a small CNF formula that relates the new variable and the subformula is appended to the result. For example, the formula $(x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (x_5 \wedge x_6 \wedge x_7)$ is transformed into:

$$
\begin{aligned}
& (y_1 \vee y_2 \vee y_3) \wedge (y_1 \equiv (x_1 \wedge x_2)) \wedge (y_2 \equiv (x_3 \wedge x_4)) \wedge (y_3 \equiv (x_5 \wedge x_6 \wedge x_7)) \\
= \quad & (y_1 \vee y_2 \vee y_3) \wedge (y_1 \Rightarrow (x_1 \wedge x_2)) \wedge ((x_1 \wedge x_2) \Rightarrow y_1) \wedge (y_2 \Rightarrow (x_3 \wedge x_4)) \wedge ((x_3 \wedge x_4) \Rightarrow y_2) \\
& \wedge (y_3 \Rightarrow (x_5 \wedge x_6 \wedge x_7)) \wedge ((x_5 \wedge x_6 \wedge x_7) \Rightarrow y_3) \\
= \quad & (y_1 \vee y_2 \vee y_3) \wedge (\neg y_1 \vee x_1) \wedge (\neg y_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee y_1) \\
& \wedge (\neg y_2 \vee x_3) \wedge (\neg y_2 \vee x_4) \wedge (\neg x_3 \vee \neg x_4 \vee y_2) \\
& \wedge (\neg y_3 \vee x_5) \wedge (\neg y_3 \vee x_6) \wedge (\neg y_3 \vee x_7) \wedge (\neg x_5 \vee \neg x_6 \vee \neg x_7 \vee y_3)
\end{aligned}
$$

with additional variables y_1, y_2 and y_3. Though the result contains more variables, it remains equi-satisfiable to the original formula, meaning that it is satisfiable if and only if the original formula is satisfiable. The variable assignment satisfying the result also satisfies the original formula after discarding the introduced variables.

Modern SAT solvers rely on a highly-optimized depth-first search with various heuristics to find a satisfying variable assignment for the given CNF formula or falsify the formula if no such

```
1: while true do
2:     if !Decide() then return SAT
3:     if !BooleanConstraintPropagation() then
4:         if !ResolveConflict() then return UNSAT
```

Figure 4.1: A typical minimal implementation of a modern SAT solver.

assignment exists. Figure 4.1 presents a typical minimal implementation, which consists of three procedures Decide(), BooleanConstraintPropagation(), and ResolveConflict().

At the beginning of each iteration, the procedure Decide() selects an unassigned variable and assigns it **true** or **false**. If all the variables have been assigned, Decide() returns **false**, indicating that a satisfying variable assignment is found and that the formula is satisfiable. The heuristics used in Decide() guides the search and has a great impact on the clauses that will be learned during the run of the SAT solver. Many SAT solvers adopts the heuristic *variable state independent decaying sum* (VSIDS) [70] for variable selection. With VSIDS, a score is associated to a variable to evaluate how active that variable is involved in the current search progress, and the most active unassigned variable is selected. The scores are halved periodically so that the solver tends to select recently active variables. A commonly used heuristic for value selection is *phase saving* [76], which always assigns a variable its last assigned value. Researchers attribute the success of VSIDS and phase saving to the search locality. Recently, a new variable selection heuristic named *learning rate branching* (LRB) [62] has been proposed. Inspired by reinforcement learning, it considers variable selection as an optimization problem and selects the variable that is likely to generate a high quantity of learned clauses.

We say a clause is *unit* if all its literals but one are assigned **false**, and the remaining literal is unassigned. The *unit propagation rule* [35] asserts that unassigned literal to be **true** for the clause to be satisfied. Such assignment to the corresponding variable is called *implication*. The procedure BooleanConstraintPropagation() applies the unit propagation rule repeatedly until no implication can be made, in which case it returns **true**, or a *conflict* occurs, i.e., all the literals in a clause are **false**, in which case it returns **false**. The performance of BooleanConstraintPropagation() depends heavily on the search for unit clauses. A lazy

data structure called *two watched literals* [70] is widely used to accelerate the search. It exploits the fact that a clause with n literals can become unit or cause a conflict only after its $n-1$ literals are assigned **false**. In other words, instead of visiting every clause containing a literal that has been assigned **false** recently, we only need to visit the clauses whose number of **false** literals goes from $n-2$ to $n-1$.

If BooleanConstraintPropagation() terminates with a conflict, the current search space does not contain any satisfying variable assignment and thus ResolveConflict() is invoked for backtracking. With the DPLL algorithm [35], the most recent decision is flipped if it has not been flipped yet, otherwise decisions and implications are revoked until an unflipped decision is reached. If all the decisions have been flipped and a conflict occurs, ResolveConflict() returns **false** and concludes that the formula is unsatisfiable. This is also called *chronological backtracking*. *Conflict-driven clause learning* (CDCL) algorithm [82, 83], which has been a standard implementation in modern SAT solvers, enhances SAT solvers with *clause learning* and *non-chronological backtracking*. It learns the reason of the conflict as a new clause by some learning scheme [83, 100] and adds it into the formula to avoid recurrence of that conflict and help prune the search space in the future. Guided by the newly learned clause, the solver backtracks to some decision that may not be the most recent unflipped one, potentially pruning a large portion of the search space. Note that the search with learning is still complete as the learned clause can be inferred from the existing clauses.

The common implementation adopts an aggressive backtracking [70], which withdraws decisions until the newly learned clause, where every literal is **false** before backtracking, contains exactly one unassigned literal, providing an implication to continue. Observing that the SAT solver may repeat lots of previous decisions and implications due to the heuristics used in Decide() to enhance search locality, we proposed a conservative strategy, named *partial backtracking* [56, 50]. Instead of simply discarding the existing decisions and the corresponding implications as aggressive backtracking does, partial backtracking tries to amend them in order to withdraw as few as possible. Backtracking is triggered only when amending causes new conflicts. As a result, some efforts spent in propagations

do not need to be repeated and the SAT solver can go deeper in a certain search space. This strategy has been implemented in our SAT solver Nigma [55, 53, 54].

Beside the three procedures above, other techniques are commonly used to improve the efficiency of SAT solvers. CNF formulas are often preprocessed and simplified [63, 88, 36, 48] before the SAT solver starts to work on them. Such simplification can also be invoked during the run of the SAT solver [49]. A randomization technique, namely *restart* [37, 45, 5, 3], periodically discards the current partial variable assignment and start over again to prevent the SAT solver from getting stuck in a complex part of the search space. Since clause learning can increase the number of clauses exponentially and then deteriorate the performance of BooleanConstraintPropagation(), clause deletion heuristics [37, 2] measure the quality of learned clauses and provide the SAT solver hints to identify and delete useless ones.

4.2.2 Bounded Model Checking

As stated in Section 3.2.2, decision diagrams have traditionally been used as the underlying representation for symbolic model checking. However, the symbolic model checking with decision diagrams has the following drawbacks: For large systems, the decision diagrams generated during model checking can become too large for current available computers. In addition, determining a proper variable ordering can be extraordinarily challenging, since it is often time-consuming or needs manual intervention. For many examples there does not exist any efficient variable ordering.

These drawbacks prompted the invention of *bounded model checking* (BMC) as a complementary technique to decision-diagram-based model checking. BMC considers witnesses of a particular bound k and generates a propositional formula that is satisfiable if and only if such a witness exists. Consequently, the model checking problem is reduced to the propositional satisfiability problem in polynomial time and can leverage the research achievements in SAT solvers. BMC was first proposed and is most successful for linear temporal logic (LTL) [7]. It can be extended to find counterexamples for ACTL or witnesses for ECTL [74], or to a larger context of $\forall\mu$-calculus [95, 71].

Though the typical BMC is SAT-based, the idea of searching for counterexamples or witnesses in a bounded model instead of the entire one also inspired decision-diagram-based techniques [97, 94].

Section 4.2.3 explains BMC for ECTL in detail. Here we give a sketch of BMC for LTL. Since checking an LTL formula is a universal model checking problem, BMC shows that the existential model checking problem for the negated formula has no solution. Intuitively, we are trying to find a counterexample to falsify the LTL formula. The formula is universally valid if we do not succeed. In the bounded semantics of LTL, we only consider a finite prefix of a path. In particular, the first $k+1$ states of a path are used to determine the validity of a formula along that path. We increase the bound k progressively, looking for longer and longer possible counterexamples. If we have considered a sufficiently large k, the bounded and unbounded semantics are equivalent.

For finite-state systems, BMC is a complete approach, meaning that it always provides a witness or concludes that no witness exists. However, in practice, it is often considered incomplete because the worst case of the sufficiently large bound could be the number of states in the system and thus too large to reach. Efforts have been made on techniques that allow to terminate the search with the conclusion that no witness exists. The *diameter* [7], which is longest shortest path between any two states, or more intuitively the maximum distance between connected states, can be the completeness threshold [28, 29] for the bound. However, the *reoccurrence diameter* [7], which is the length of the longest path where all states are different, drew more attention for its possibility to be formulated in SAT, while the diameter is conjectured to need *quantified boolean formula* (QBF).

4.2.3 Bounded Model Checking for Existential CTL

SAT-based BMC for ACTL formulas was proposed in [74], and later improved with a more efficient translation to propositional formulas [98, 18]. Similar to BMC for LTL, this approach also turns a universal model checking problem into an existential one, by negating the ACTL formula, and actually searches for a witness for the corresponding ECTL formula over a bounded model. The main difference is that this approach represents a witness as a set of symbolic paths, each of which has length k, due to the branching time nature of CTL. For example, a witness for $\mathsf{EG}(\mathsf{EF}a)$

contains a lasso-shaped loop of length k where $\mathsf{EF}a$ holds in every state, and, for each state on that loop, a path of length k showing that a state satisfying a can be reached from it. The bound k does not describe the size of a potential witness, but the length of each individual path demonstrating satisfaction of a subformula in a state. In the final witness we find, it is possible that the postfixes of some paths are not necessary and that a path of length less than k may be sufficient to demonstrate a subformula in a state. But the longest sufficient path has length k.

We define the bounded semantics of ECTL in Section 4.2.3.1 and demonstrate how to translate an ECTL model checking problem into a propositional formula in Section 4.2.3.2.

4.2.3.1 Bounded Semantics of Existential CTL

Given a model $M = (\mathcal{S}, \mathcal{S}_{init}, \mathcal{N}, \mathcal{A}, \mathcal{L})$ and $k \in \mathbb{N}$, a *k-path*, or a path of length k, is a finite sequence $\pi = [\![s_0, ..., s_k]\!]$ of $k+1$ states such that $(s_i, s_{i+1}) \in \mathcal{N}$ for any $i \in \{0, ..., k-1\}$. Let $\pi[i]$ denote s_i, the i-th state in π. A k-path, though finite, can still represent an infinite path if the last state is the same as any of the previous states. We define a function $loop(\pi)$ to determine if a k-path π is a loop:

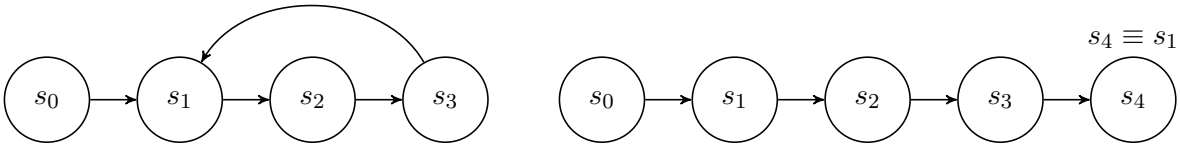
$$loop(\pi) = \{i \in \{0, ..., k-1\} \mid \pi[k] \equiv \pi[i]\},$$

thus $loop(\pi) \neq \emptyset$ iff π is a loop.

In our definition, a loop is thus a k-path containing at least two states [61]. This is slightly different from the notation in [7, 74], which explicitly requires a back loop from the last state to some state on the path: $\{i \in \{0, ..., k\} \mid (\pi[k], \pi[i]) \in \mathcal{N}\}$. Figure 4.2 illustrates the two ways of thinking about, and defining, the same loop. We choose this notation because encoding state equivalence is often simpler and more compact than encoding a transition relation step. Our notation requires $k \geq 1$, which is then assumed in the rest of the chapter.

The *k-model* of M is a tuple $M_k = (\mathcal{S}, \mathcal{S}_{init}, \mathcal{P}_k, \mathcal{A}, \mathcal{L})$, where $\mathcal{P}_k$ is the set of all the k-paths in M. The bounded semantics is defined over a k-model M_k.

Definition 4.2.1 (Bounded semantics of ECTL)**.** *Let M_k be the k-model of a model M, s a state of M, a an atomic proposition, and φ, ρ ECTL formulas. The conditions for s in M_k to satisfy φ,*



(a) The loop shape in [7, 74] (b) The loop shape in this chapter

Figure 4.2: The two kinds of loop shapes

written $s \models_k \varphi$ *(M_k is omitted for brevity), are defined as follows:*

$$s \models_k a \Leftrightarrow a \in \mathcal{L}(s)$$

$$s \models_k \neg a \Leftrightarrow a \notin \mathcal{L}(s)$$

$$s \models_k \varphi \wedge \rho \Leftrightarrow (s \models_k \varphi) \wedge (s \models_k \rho)$$

$$s \models_k \varphi \vee \rho \Leftrightarrow (s \models_k \varphi) \vee (s \models_k \rho)$$

$$s \models_k \mathsf{EX}\varphi \Leftrightarrow \exists \pi \in \mathcal{P}_k, (\pi[0] \equiv s) \wedge (\pi[1] \models_k \varphi)$$

$$s \models_k \mathsf{E}(\varphi \mathsf{U} \rho) \Leftrightarrow \exists \pi \in \mathcal{P}_k, (\pi[0] \equiv s) \wedge (\exists i \in \{0, ..., k\}, (\pi[i] \models_k \rho) \wedge (\forall j \in \{0, ..., i-1\}, \pi[j] \models_k \varphi))$$

$$s \models_k \mathsf{EG}\varphi \Leftrightarrow \exists \pi \in \mathcal{P}_k, (\pi[0] \equiv s) \wedge (loop(\pi) \neq \emptyset) \wedge (\forall i \in \{0, ..., k\}, \pi[i] \models_k \varphi).$$

Theorem 4.2.1. *Let M be a model, $\mathcal{R}$ the reachable states of M, s a state of M, and φ an ECTL formula. Then, $s \models \varphi$ iff $s \models_k \varphi$ for some $k \in \{1, ..., |\mathcal{R}|\}$.*

Recall that we assume $\mathcal{S}_{init}$ contains a single state s_{init}, $\mathcal{S}_{init} = \{s_{init}\}$, to avoid distinguishing existential and universal validity.

Definition 4.2.2. *An ECTL formula φ is valid in a k-model $M_k = (\mathcal{S}, \mathcal{S}_{init}, \mathcal{P}_k, \mathcal{A}, \mathcal{L})$, written $M \models_k \varphi$, iff $s_{init} \models_k \varphi$.*

Theorem 4.2.2. *Given an ECTL formula φ, $M \models \varphi$ iff $M \models_k \varphi$ for some $k \in \{1, ..., |\mathcal{R}|\}$.*

Based on Theorem 4.2.2, an ECTL model checking problem is reduced to an ECTL BMC problem: the unbounded and bounded semantics are equivalent for a sufficiently large bound.

4.2.3.2 Translation into a Proposition Formula

A witness is represented as the result of gluing a set of k-paths, whose size is determined by the shape of the maximum possible witness, for the given ECTL formula and the value of k. This number is given by the following function $\mathcal{F}_k$:

Definition 4.2.3. *Function $\mathcal{F}_k : ECTL \to \mathbb{N}$ is defined as follows:*

$$\mathcal{F}_k(a) = 0, \text{ where } a \in \mathcal{A}$$

$$\mathcal{F}_k(\neg a) = 0, \text{ where } a \in \mathcal{A}$$

$$\mathcal{F}_k(\varphi \wedge \rho) = \mathcal{F}_k(\varphi) + \mathcal{F}_k(\rho)$$

$$\mathcal{F}_k(\varphi \vee \rho) = \max(\mathcal{F}_k(\varphi), \mathcal{F}_k(\rho))$$

$$\mathcal{F}_k(\mathsf{EX}\varphi) = \mathcal{F}_k(\varphi) + 1$$

$$\mathcal{F}_k(\mathsf{E}(\varphi \mathsf{U} \rho)) = k \cdot \mathcal{F}_k(\varphi) + \mathcal{F}_k(\rho) + 1$$

$$\mathcal{F}_k(\mathsf{EG}\varphi) = k \cdot \mathcal{F}_k(\varphi) + 1,$$

where $\mathcal{F}_k(\mathsf{EG}\varphi)$ is slightly different from that in [74, 98, 18] because of our loop notation.

For example, given a bound k, the maximum witness to show $\mathsf{E}(\varphi \mathsf{U} \rho)$ consists of a k-path $[\![s_0, s_1, ..., s_k]\!]$, $\mathcal{F}_k(\rho)$ k-paths to show $s_k \models \rho$, and, for each $i \in \{0, ..., k-1\}$, $\mathcal{F}_k(\varphi)$ k-paths to show $s_i \models \varphi$. The number of k-paths is $k \cdot \mathcal{F}_k(\varphi) + \mathcal{F}_k(\rho) + 1$ in total.

Symbolically, a state is represented as a vector of boolean variables. Let π_i denote the i-th symbolic k-path, which is a sequence of $k+1$ symbolic states. Checking an ECTL formula φ over a bounded model M_k is then reduced to checking the satisfiability of a propositional formula $[M, \varphi]_k = [M_k]_\varphi \wedge [\varphi]_k$, where:

- $[M_k]_\varphi$ is a propositional formula that enforces the $\mathcal{F}_k(\varphi)$ state sequences to be valid k-paths and $\pi_0[0]$ to be an initial state:

$$[M_k]_\varphi = I(\pi_0[0]) \wedge \bigwedge_{i=0}^{\mathcal{F}_k(\varphi)-1} \bigwedge_{j=0}^{k-1} \mathcal{N}(\pi_i[j], \pi_i[j+1]),$$

  where $I(s)$ iff $s \equiv s_{init} \in \mathcal{S}_{init}$ and $\mathcal{N}(s, s')$ iff $(s, s') \in \mathcal{N}$.

- $[\varphi]_k = [\varphi, \pi_0[0]]_k^0$ is a propositional formula that assembles k-paths and enforces φ or subformulas of φ on each corresponding k-path; specifically, $[\varphi, s]_k^i$ enforces φ on the i-th k-path, where the first state of that k-path must be equivalent to s (unless φ is a pure propositional formula):

$$
\begin{aligned}
[a, s]_k^i &= a(s), \text{ where } a \in \mathcal{A} \\
[\neg a, s]_k^i &= \neg a(s), \text{ where } a \in \mathcal{A} \\
[\varphi \wedge \rho, s]_k^i &= [\varphi, s]_k^i \wedge [\rho, s]_k^{i+\mathcal{F}_k(\varphi)} \\
[\varphi \vee \rho, s]_k^i &= [\varphi, s]_k^i \vee [\rho, s]_k^i \\
[\mathsf{EX}\varphi, s]_k^i &= (s \equiv \pi_i[0]) \wedge [\varphi, \pi_i[1]]_k^{i+1} \\
[\mathsf{E}(\varphi \mathsf{U} \rho), s]_k^i &= (s \equiv \pi_i[0]) \wedge \bigvee_{j=0}^{k} \left([\rho, \pi_i[j]]_k^{i+1} \wedge \bigwedge_{t=0}^{j-1} [\varphi, \pi_i[t]]_k^{i+1+\mathcal{F}_k(\rho)+t\cdot\mathcal{F}_k(\varphi)} \right) \\
[\mathsf{EG}\varphi, s]_k^i &= (s \equiv \pi_i[0]) \wedge \bigvee_{j=0}^{k-1} (\pi_i[k] \equiv \pi_i[j]) \wedge \bigwedge_{j=0}^{k-1} [\varphi, \pi_i[j]]_k^{i+1+j\cdot\mathcal{F}_k(\varphi)},
\end{aligned}
$$

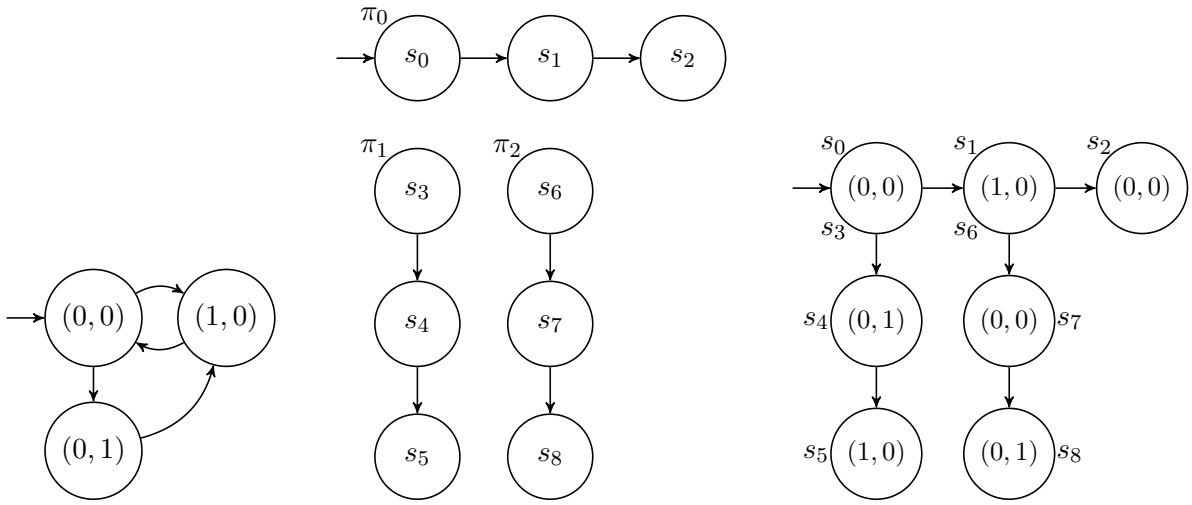
  where, again, $[\mathsf{EG}\varphi, s]_k^i$ is slightly different from that in [74, 98, 18].

For example, the interpretation of $[\mathsf{E}(\varphi \mathsf{U} \rho), s]_k^i$ is as follows: First, the first state of π_i is equivalent to the given state s. Then, for each $j \in \{0, ..., k\}$, we start from π_{i+1} to search for a witness for $\pi_i[j] \models \rho$, consisting of $\mathcal{F}_k(\rho)$ k-paths, and from $\pi_{i+1+\mathcal{F}_k(\rho)+t\cdot\mathcal{F}_k(\varphi)}$ to search for a witness for $\pi_i[t] \models \varphi$, consisting of $\mathcal{F}_k(\varphi)$ k-paths, for each $t \in \{0, ..., j-1\}$.

Consider the Kripke structure in Figure 4.3(a), where each state is a Boolean vector (x, y). Suppose that we are verifying the formula $\mathsf{EG}(\mathsf{EF}(y = 1))$ and translating its bounded semantics into a propositional formula when $k = 2$. Therefore, the number of k-paths needed is:

$$
\begin{aligned}
\mathcal{F}_2(\mathsf{EG}(\mathsf{EF}(y = 1))) &= 2 \cdot \mathcal{F}_2(\mathsf{EF}(y = 1)) + 1 \\
&= 2 \cdot (2 \cdot \mathcal{F}_2(\mathbf{true}) + \mathcal{F}_2(y = 1) + 1) + 1 \\
&= 3
\end{aligned}
$$

and the number of symbolic states is: $3 \cdot (k+1) = 9$. In particular, $\pi_0 = [\![s_0, s_1, s_2]\!]$, $\pi_1 = [\![s_3, s_4, s_5]\!]$, $\pi_2 = [\![s_6, s_7, s_8]\!]$, as shown in Figure 4.3(b). Let $s_i = (x_i, y_i)$.



(a) A Kripke structure. (b) k-paths and symbolic states. (c) A witness for $\mathsf{EG}(\mathsf{EF}(y = 1))$.

Figure 4.3: An example for translating a BMC problem.

First, we generate $[M_2]_{\mathsf{EG}(\mathsf{EF}(y=1))}$:

$$
\begin{aligned}
[M_2]_{\mathsf{EG}(\mathsf{EF}(y=1))} = {} & (\neg x_0 \wedge \neg y_0) && (s_0 \text{ is the initial state}) \\
& \wedge \mathcal{N}(s_0, s_1) \wedge \mathcal{N}(s_1, s_2) && (\pi_0 \text{ is a valid path}) \\
& \wedge \mathcal{N}(s_3, s_4) \wedge \mathcal{N}(s_4, s_5) && (\pi_1 \text{ is a valid path}) \\
& \wedge \mathcal{N}(s_6, s_7) \wedge \mathcal{N}(s_7, s_8) && (\pi_2 \text{ is a valid path})
\end{aligned}
$$

where:

$$
\begin{aligned}
\mathcal{N}(s_i, s_j) = {} & (\neg x_i \wedge \neg y_i \wedge x_j \wedge \neg y_j) && ((0,0) \to (1,0)) \\
& \vee (x_i \wedge \neg y_i \wedge \neg x_j \wedge \neg y_j) && ((1,0) \to (0,0)) \\
& \vee (\neg x_i \wedge \neg y_i \wedge \neg x_j \wedge y_j) && ((0,0) \to (0,1)) \\
& \vee (\neg x_i \wedge y_i \wedge x_j \wedge \neg y_j) && ((0,1) \to (1,0))
\end{aligned}
$$

Then we generate $[\mathsf{EG}(\mathsf{EF}(y = 1))]_2$:

$$
\begin{aligned}
[\mathsf{EG}(\mathsf{EF}(y = 1))]_2 &= [\mathsf{EG}(\mathsf{EF}(y = 1)), s_0]_2^0 \\
&= (s_0 \equiv s_0) \wedge \big((s_2 \equiv s_0) \vee (s_2 \equiv s_1)\big) \wedge [\mathsf{EF}(y = 1), s_0]_2^1 \wedge [\mathsf{EF}(y = 1), s_1]_2^2
\end{aligned}
$$

where:

$$[\mathsf{EF}(y=1), s_0]_2^1 = (s_0 \equiv s_3) \wedge \left([(y=1), s_3]_2^2 \vee [(y=1), s_4]_2^2 \vee [(y=1), s_5]_2^2\right)$$
$$= (s_0 \equiv s_3) \wedge (y_3 \vee y_4 \vee y_5)$$

$$[\mathsf{EF}(y=1), s_1]_2^2 = (s_1 \equiv s_6) \wedge \left([(y=1), s_6]_2^3 \vee [(y=1), s_7]_2^3 \vee [(y=1), s_8]_2^3\right)$$
$$= (s_1 \equiv s_6) \wedge (y_6 \vee y_7 \vee y_8)$$

$$(s_i \equiv s_j) = (x_i \equiv x_j) \wedge (y_i \equiv y_j)$$
$$= (x_i \Rightarrow x_j) \wedge (x_j \Rightarrow x_i) \wedge (y_i \Rightarrow y_j) \wedge (y_j \Rightarrow y_i)$$
$$= (\neg x_i \vee x_j) \wedge (x_i \vee \neg x_j) \wedge (\neg y_i \vee y_j) \wedge (y_i \vee \neg y_j)$$

Then $[M, \mathsf{EG}(\mathsf{EF}(y=1))]_2$, the conjunction of $[M_2]_{\mathsf{EG}(\mathsf{EF}(y=1))}$ and $[\mathsf{EG}(\mathsf{EF}(y=1))]_2$, is transformed into CNF. A SAT solver can find a satisfying assignment for it, from which we can build a witness, for example, the one in Figure 4.3(c). In this witness, s_0 and s_3, s_1 and s_6 coincide. Note that s_5 is actually not necessary and removing it does not affect the validity of the witness that is finally presented.

We refer to the translation above as the Classic approach.

Theorem 4.2.3. *Let M be a model, and φ an ECTL formula. $M \models_k \varphi$ iff $[M, \varphi]_k$ is satisfiable.*

Theorem 4.2.4. *$M \models \varphi$ iff for some $k \in \{1, ..., |\mathcal{R}|\}$, $[M, \varphi]_k$ is satisfiable.*

In practice, it is often the case that, if $M \models \varphi$, there exists a small k such that $M \models_k \varphi$, i.e., a "shallow" witness exists. This is the reason why BMC is often very efficient in error detection. The deeper the witness is, the less advantage BMC has.

4.3 Improved Translation of Bounded Semantics

We now give an example to explain how we improve on Classic. Given an ECTL formula, the goal of BMC is to find a witness demonstrating the satisfaction. Figure 4.4(a) can be viewed

as a witness for $\mathsf{EG}(\mathsf{EF}a)$ when $k = 3$, consisting of two k-paths, $[\![1, 2, 3, 2]\!]$ and $[\![3, 4, 5, *]\!]$, where $*$ can be any valid state, because we can extract Figure 4.4(b) from it. By *reusing* paths, we can represent a witness in such a compact form. For example, the witness for $\mathsf{EF}a$ in state 2 is built by concatenating state 2 to the witness for $\mathsf{EF}a$ in state 3. BMC can benefit from this compact form, since the minimum k to find a witness is determined by the longest subpath (not counting $*$) in it. We can find the witness in Figure 4.4(a) when $k = 3$, since the longest subpath is $[\![1, 2, 3, 2]\!]$. Without reusing, we must have $k = 4$, since the longest subpath is now $[\![1, 2, 3, 4, 5]\!]$, the witness for $\mathsf{EF}a$ in state 1, and use five k-paths to find that witness (in a different form) as shown in Figure 4.4(c): $[\![1, 2, 3, 2, 3]\!]$, $[\![1, 2, 3, 4, 5]\!]$, $[\![2, 3, 4, 5, *]\!]$, $[\![3, 4, 5, *, *]\!]$, and $[\![2, 3, 4, 5, *]\!]$.

Given two states s and s' such that $(s, s') \in \mathcal{N}$, if $\mathsf{E}(\varphi \mathsf{U} \rho)$ (or $\mathsf{EG}\varphi$) holds in s', and φ holds in s, then $\mathsf{E}(\varphi \mathsf{U} \rho)$ (or $\mathsf{EG}\varphi$) also holds in s. This is due to the inductive definitions of CTL temporal operators, and can be exploited to improve the translation of bounded semantics. To enforce $\mathsf{E}(\varphi \mathsf{U} \rho)$ to hold in every state along a finite path, we only need to enforce $\mathsf{E}(\varphi \mathsf{U} \rho)$ to hold in the last state, and $\varphi \vee \rho$ in any previous state. Similarly, in the case of $\mathsf{EG}\varphi$, we only need to enforce $\mathsf{EG}\varphi$ to hold in the last state, while φ is enough in any previous state. This potentially reduces the number of necessary k-paths, thus the size of the resulting propositional formula, for symbolic representation of a witness in the bounded model.

We then define an auxiliary function $\mu : ECTL \to ECTL$ providing the *sufficient predecessor formula* to be enforced in "any previous state":

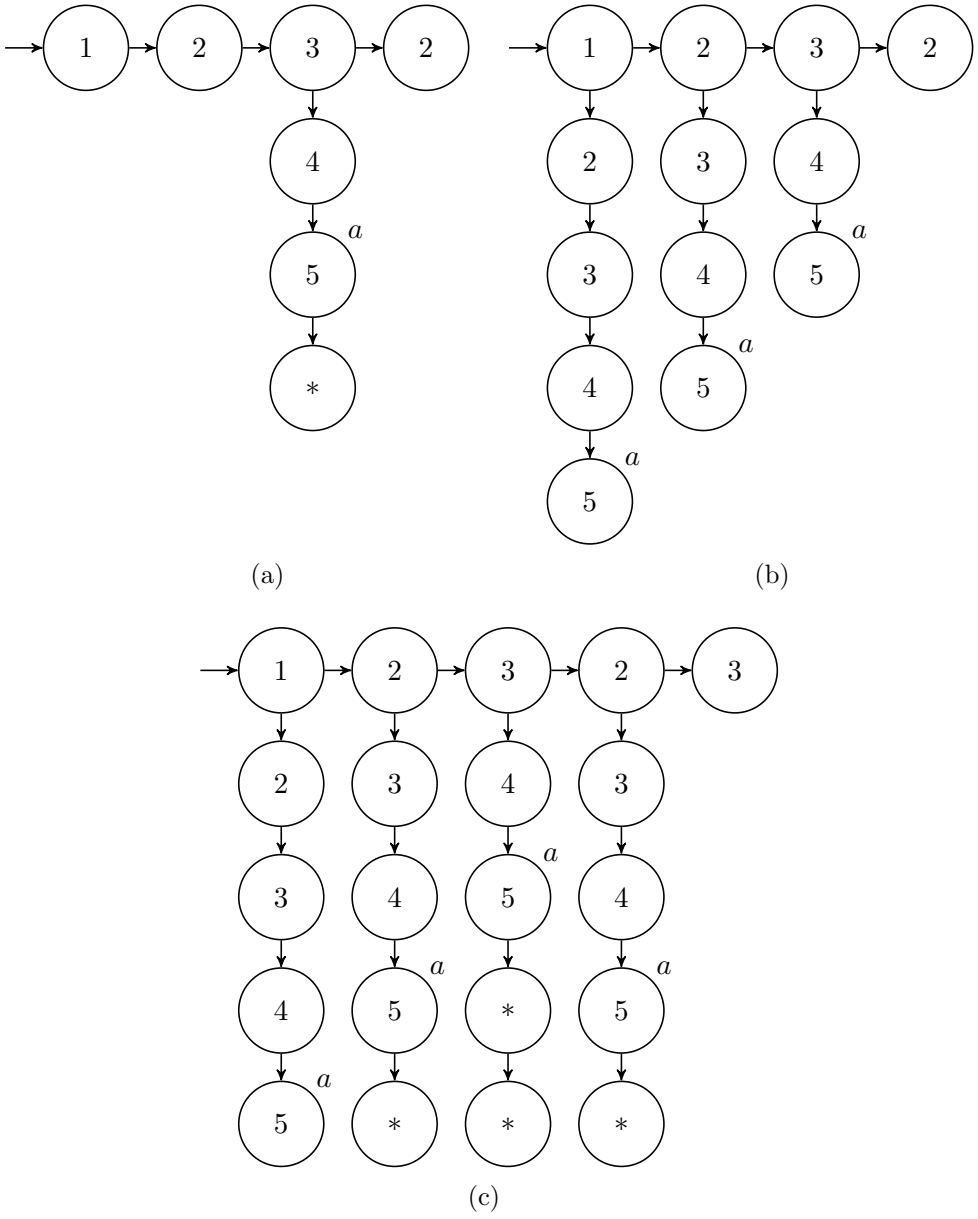


Figure 4.4: Different forms of witnesses for $\mathsf{EG}(\mathsf{EF}a)$.

Definition 4.3.1. *Function $\mu : ECTL \to ECTL$ is defined as follows:*

$$\begin{aligned}
\mu(a) &= a, \ \textit{where } a \in \mathcal{A} \\
\mu(\neg a) &= \neg a, \ \textit{where } a \in \mathcal{A} \\
\mu(\varphi \wedge \rho) &= \mu(\varphi) \wedge \mu(\rho) \\
\mu(\varphi \vee \rho) &= \varphi \vee \rho \\
\mu(\mathsf{EX}\varphi) &= \mathsf{EX}\varphi \\
\mu(\mathsf{E}(\varphi \mathsf{U} \rho)) &= \varphi \vee \rho \\
\mu(\mathsf{EG}\varphi) &= \mu(\varphi)\,.
\end{aligned}$$

Theorem 4.3.1. *Let $M = (\mathcal{S}, \mathcal{S}_{init}, \mathcal{N}, \mathcal{A}, \mathcal{L})$ be a model, s, s' states of M such that $(s, s') \in \mathcal{N}$, and φ an ECTL formula. If $s' \models \varphi$ and $s \models \mu(\varphi)$, then $s \models \varphi$.*

Proof. We only need to prove correctness in the cases of conjunction, EU, and EG.

- If $s' \models \mathsf{E}(\varphi \mathsf{U} \rho)$ and $s \models \varphi \vee \rho$, then $s \models \mathsf{E}(\varphi \mathsf{U} \rho)$:
  If $s \models \rho$, it is trivial to see that $s \models \mathsf{E}(\varphi \mathsf{U} \rho)$. If $s \models \varphi$, since s has a successor $s' \models \mathsf{E}(\varphi \mathsf{U} \rho)$, then $s \models \mathsf{E}(\varphi \mathsf{U} \rho)$ by the definition of EU.

The proof for conjunction and EG is based on induction on the structure of the ECTL formula. The basis is that $\mu(a) = a$, whose correctness is trivial.

- If $s' \models \varphi \wedge \rho$ and $s \models \mu(\varphi) \wedge \mu(\rho)$, then $s \models \varphi \wedge \rho$:
  Assume that, if $s' \models \varphi$ and $s \models \mu(\varphi)$ then $s \models \varphi$, and that, if $s' \models \rho$ and $s \models \mu(\rho)$ then $s \models \rho$. By these assumptions, we have $s \models \varphi$ and $s \models \rho$, thus $s \models \varphi \wedge \rho$.

- If $s' \models \mathsf{EG}\varphi$ and $s \models \mu(\varphi)$, then $s \models \mathsf{EG}\varphi$:
  Assume that, if $s' \models \varphi$ and $s \models \mu(\varphi)$ then $s \models \varphi$. Since $s' \models \mathsf{EG}\varphi$, we have $s' \models \varphi$, thus $s \models \varphi$ by assumption; since s has a successor $s' \models \mathsf{EG}\varphi$, $s \models \mathsf{EG}\varphi$ by the definition of EG.

□

It is worthwhile to emphasize that Theorem 4.3.1 does not hold if we define $\mu(\varphi \vee \rho)$ as $\mu(\varphi) \vee \mu(\rho)$. Consider the simple example where $\varphi = \mathsf{EG}a$ and $\rho = \mathsf{EG}b$, in Figure 4.5. $(\mathsf{EG}a) \vee (\mathsf{EG}b)$ does not hold in s, because $[\![s, s', s']\!]$, obtained through path reuse, is not a witness for $s \models (\mathsf{EG}a) \vee (\mathsf{EG}b)$, even though $a \vee b$ holds in s and (s', s') witnesses $(\mathsf{EG}a) \vee (\mathsf{EG}b)$ in s'.

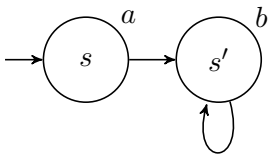


Figure 4.5: An example showing that $\mu(\varphi \vee \rho) \neq \mu(\varphi) \vee \mu(\rho)$.

According to Theorem 4.3.1, given a finite path $[\![s_0, ..., s_n]\!]$, to enforce an ECTL formula φ in s_i for any $i \in \{0, ..., n\}$, we enforce $s_n \models \varphi$ but just $s_i \models \mu(\varphi)$, for $i \in \{0, ..., n-1\}$, which usually simplifies the formula to be enforced in s_i.

We define function $\mathcal{G}_k$, as an improvement of $\mathcal{F}_k$, giving the potentially smaller number of k-paths needed to check an ECTL formula in a bounded model. $\mathcal{G}_k$ differs from $\mathcal{F}_k$ only in the case of EU and EG:

Definition 4.3.2. *Function $\mathcal{G}_k : ECTL \to \mathbb{N}$ is defined as follows:*

$$\mathcal{G}_k(a) = 0, \text{ where } a \in \mathcal{A}$$

$$\mathcal{G}_k(\neg a) = 0, \text{ where } a \in \mathcal{A}$$

$$\mathcal{G}_k(\varphi \wedge \rho) = \mathcal{G}_k(\varphi) + \mathcal{G}_k(\rho)$$

$$\mathcal{G}_k(\varphi \vee \rho) = \max(\mathcal{G}_k(\varphi), \mathcal{G}_k(\rho))$$

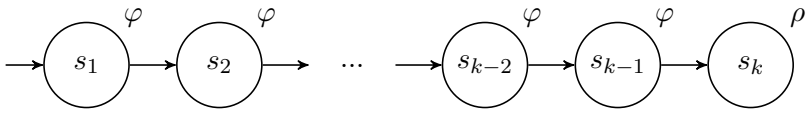
$$\mathcal{G}_k(\mathsf{EX}\varphi) = \mathcal{G}_k(\varphi) + 1$$

$$\mathcal{G}_k(\mathsf{E}(\varphi \mathsf{U} \rho)) = (k-1) \cdot \mathcal{G}_k(\mu(\varphi)) + \mathcal{G}_k(\varphi) + \mathcal{G}_k(\rho) + 1$$

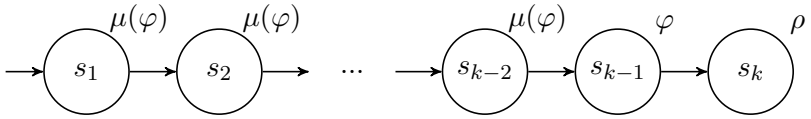
$$\mathcal{G}_k(\mathsf{EG}\varphi) = (k-1) \cdot \mathcal{G}_k(\mu(\varphi)) + \mathcal{G}_k(\varphi) + 1\,.$$

For example, given a bound k, the maximum witness to show $\mathsf{E}(\varphi \mathsf{U} \rho)$, according to our approach, consists of a k-path $[\![s_0, ..., s_k]\!]$, $\mathcal{G}_k(\rho)$ k-paths to show $s_k \models \rho$, $\mathcal{G}_k(\varphi)$ paths to show $s_{k-1} \models \varphi$, and,

for each $i \in \{0, ..., k-2\}$, $\mathcal{G}_k(\mu(\varphi))$ k-paths to show $s_i \models \mu(\varphi)$, since $s_i \models \varphi$ can be inferred. The number of k-paths is $(k-1) \cdot \mathcal{G}_k(\mu(\varphi)) + \mathcal{G}_k(\varphi) + \mathcal{G}(\rho) + 1$ in total. Figure 4.6 illustrates the difference in the translation of $\mathsf{E}(\varphi \mathsf{U} \rho)$ with and without path reuse.



(a) Without path reuse



(b) With path reuse

Figure 4.6: Translation of $\mathsf{E}(\varphi \mathsf{U} \rho)$ with and without path reuse.

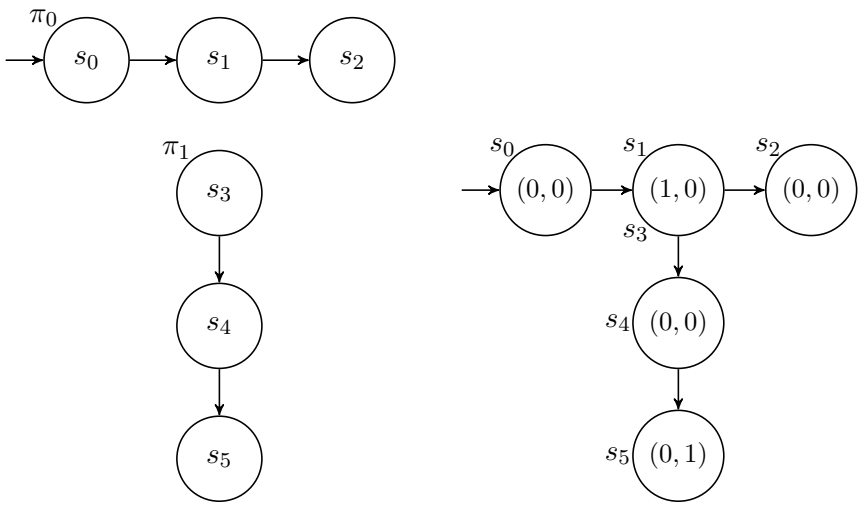
Theorem 4.3.2. *Given an ECTL formula φ, $\mathcal{G}_k(\mu(\varphi)) \leq \mathcal{G}_k(\varphi)$.*

Theorem 4.3.3. *Given an ECTL formula φ, $\mathcal{G}_k(\varphi) \leq \mathcal{F}_k(\varphi)$.*

Then, we update the translation of conjunctive, EU, and EG formulas to propositional formulas (while we just replace $\mathcal{F}_k$ with $\mathcal{G}_k$ for the remaining formulas):

$$
\begin{aligned}
[\varphi \wedge \rho, s]_k^i &= [\mu(\varphi), s]_k^i \wedge [\mu(\rho), s]_k^{i+\mathcal{G}_k(\mu(\varphi))} \\
[\mathsf{E}(\varphi \mathsf{U} \rho), s]_k^i &= (s \equiv \pi_i[0]) \wedge \Bigg([\rho, \pi_i[0]]_k^{i+1} \vee \bigvee_{j=1}^{k} \left([\rho, \pi_i[j]]_k^{i+1} \wedge [\varphi, \pi_i[j-1]]_k^{i+1+\mathcal{G}_k(\rho)} \right. \\
&\qquad \left. \wedge \bigwedge_{t=0}^{j-2} [\mu(\varphi), \pi_i[t]]_k^{i+1+\mathcal{G}_k(\rho)+\mathcal{G}_k(\varphi)+t \cdot \mathcal{G}_k(\mu(\varphi))} \right) \Bigg) \\
[\mathsf{EG}\varphi, s]_k^i &= (s \equiv \pi_i[0]) \wedge \bigvee_{j=0}^{k-1} (\pi_i[k] \equiv \pi_i[j]) \\
&\qquad \wedge [\varphi, \pi_i[k-1]]_k^{i+1} \wedge \bigwedge_{j=0}^{k-2} [\mu(\varphi), \pi_i[j]]_k^{i+1+\mathcal{G}_k(\varphi)+j \cdot \mathcal{G}_k(\mu(\varphi))}.
\end{aligned}
$$

Now we consider the Kripke structure in Figure 4.3(a) again and generate a propositional formula encoding the bounded semantics of $\mathsf{EG}(\mathsf{EF}(y = 1))$ when $k = 2$ using the improved translation.



(a) k-paths and symbolic states. (b) A witness for $\mathsf{EGEF}(y=1)$.

Figure 4.7: An example for translating a BMC problem with path reuse.

With path reuse, the number of k-paths needed is:

$$\begin{aligned}\mathcal{G}_2(\mathsf{EG}(\mathsf{EF}(y=1))) &= \mathcal{G}_2(\mu(\mathsf{EF}(y=1))) + \mathcal{G}_2(\mathsf{EF}(y=1)) + 1 \\ &= \mathcal{G}_2(\mathbf{true}) + \mathcal{G}_2(\mu(\mathbf{true})) + \mathcal{G}_2(\mathbf{true}) + \mathcal{G}_2(y=1) + 1 + 1 \\ &= 2\end{aligned}$$

and the number of symbolic states is: $2\cdot(2+1) = 6$. In particular, $\pi_0 = [\![s_0, s_1, s_2]\!]$, $\pi_2 = [\![s_3, s_4, s_5]\!]$, as shown in Figure 4.7(a).

As before, we start with $[M_2]_{\mathsf{EG}(\mathsf{EF}(y=1))}$. Due to fewer paths, $[M_2]_{\mathsf{EG}(\mathsf{EF}(y=1))}$ becomes more succinct than that without path reuse:

$$\begin{aligned}[M_2]_{\mathsf{EG}(\mathsf{EF}(y=1))} = (\neg x_0 \wedge \neg y_0) \quad & (s_0 \text{ is the initial state}) \\ \wedge\, \mathcal{N}(s_0, s_1) \wedge \mathcal{N}(s_1, s_2) \quad & (\pi_0 \text{ is a valid path}) \\ \wedge\, \mathcal{N}(s_3, s_4) \wedge \mathcal{N}(s_4, s_5) \quad & (\pi_1 \text{ is a valid path})\end{aligned}$$

Then $[\mathsf{EG}(\mathsf{EF}(y=1))]_2$ is generated as follows:

$$\begin{aligned}
[\mathsf{EG}(\mathsf{EF}(y=1))]_2 &= [\mathsf{EG}(\mathsf{EF}(y=1)), s_0]_2^0 \\
&= (s_0 \equiv s_0) \wedge \big((s_2 \equiv s_0) \vee (s_2 \equiv s_1)\big) \wedge [\mathsf{EF}(y=1), s_1]_2^1 \wedge [\mu(\mathsf{EF}(y=1)), s_0]_2^2 \\
&= (s_0 \equiv s_0) \wedge \big((s_2 \equiv s_0) \vee (s_2 \equiv s_1)\big) \wedge [\mathsf{EF}(y=1), s_1]_2^1 \wedge [\mathbf{true}, s_0]_2^2 \\
&= (s_0 \equiv s_0) \wedge \big((s_2 \equiv s_0) \vee (s_2 \equiv s_1)\big) \wedge [\mathsf{EF}(y=1), s_1]_2^1
\end{aligned}$$

where:

$$\begin{aligned}
[\mathsf{EF}(y=1), s_1]_2^1 &= (s_1 \equiv s_3) \wedge \big([(y=1), s_3]_2^3 \vee [(y=1), s_4]_2^3 \vee [(y=1), s_5]_2^3\big) \\
&= (s_1 \equiv s_3) \wedge (y_3 \vee y_4 \vee y_5)
\end{aligned}$$

A witness in a compact form in Figure 4.7(b) can be built from the satisfying assignment for the conjunction of $[M_2]_{\mathsf{EG}(\mathsf{EF}(y=1))}$ and $[\mathsf{EG}(\mathsf{EF}(y=1))]_2$. s_1 and s_3 coincide in it. The subwitness for $s_0 \models \mathsf{EF}(y=1)$ is $[\![(0,0),(1,0),(0,0),(0,1)]\!]$, obtained by reusing the subwitness for $s_1 \models \mathsf{EF}(y=1)$, i.e., $[\![(1,0),(0,0),(0,1)]\!]$.

We refer to this new translation with path reuse as the Reuse approach.

The templates of ACTL and ECTL formulas guaranteeing that their instances have linear counterexamples and witnesses (if they exist) have been studied in [15, 96]. For formulas instantiated from linear ECTL templates, $\mathcal{G}_k(\varphi) = \mathcal{F}_k(\varphi)$, i.e., Reuse generates exactly the same propositional formula as Classic. For some ECTL formulas whose general form of witnesses is not linear (e.g., $\mathsf{EG}(\mathsf{EX}a)$, $(\mathsf{EG}a) \vee (\mathsf{EG}b)$), the two approaches may also generate the same formulas. A complete template of ECTL formulas for which $\mathcal{G}_k(\varphi) < \mathcal{F}_k(\varphi)$ is still unclear.

4.4 Comparison of the Two Translation Approaches

In this section, we compare Classic and Reuse with respect to the minimum bound needed to find a witness (Section 4.4.1) and the complexity of propositional formulas (Section 4.4.2).

4.4.1 Minimum Bound to Find a Witness

We already saw an example where REUSE finds a witness using a smaller k than CLASSIC, at the beginning of Section 4.3. Let k_{min}^{CLASSIC} and k_{min}^{REUSE} denote the minimum bounds for which CLASSIC and REUSE find a witness for a particular formula, respectively.

Theorem 4.4.1. *Given an ECTL formula φ holding in a model M, $k_{min}^{\text{REUSE}} \leq k_{min}^{\text{CLASSIC}}$.*

Proof. If CLASSIC finds a witness, REUSE can always find a witness in the compact form, as a substructure of the witness found by CLASSIC, thus using the same k. For example, suppose CLASSIC finds the witness for $\mathsf{E}((\mathsf{EF}a)\mathsf{U}b)$ shown in Figure 4.8, using $k = 3$. Its substructure (shown in the dashed box) is a witness REUSE can also find. This implies that k_{min}^{REUSE} is never greater than k_{min}^{CLASSIC}. □

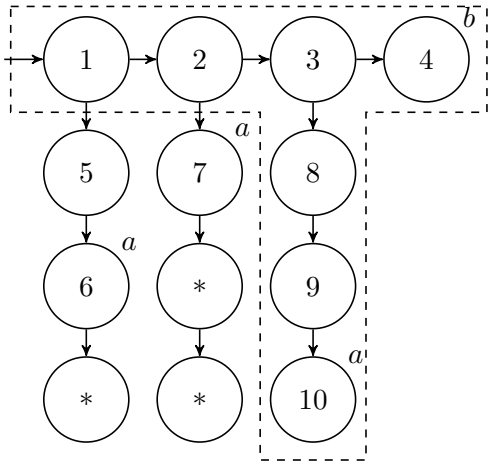


Figure 4.8: A witness for $\mathsf{E}((\mathsf{EF}a)\mathsf{U}b)$.

Theorem 4.4.2. *Given an ECTL formula φ holding in a model M, k_{min}^{CLASSIC} can be as large as $2k_{min}^{\text{REUSE}} - 1$.*

Proof. To prove this theorem, it suffices to consider the ECTL formula $\varphi = \mathsf{E}(\mathsf{E}(a\mathsf{U}b)\mathsf{U}c)$ and the model shown in Figure 4.9. REUSE seeks two k-paths, one path π where some $\pi[j]$ satisfies c, $\pi[j-1]$ satisfies $\mathsf{E}(a\mathsf{U}b)$, and $\pi[0], ..., \pi[j-2]$ satisfy $\mu(\mathsf{E}(a\mathsf{U}b)) = a \vee b$, and another path σ where $\sigma[0]$ coincides with $\pi[j-1]$, some $\sigma[l]$ satisfies b, and $\sigma[0], ..., \sigma[l-1]$ satisfy a. For the model in

Figure 4.9, these two paths correspond to $[\![s_0, ..., s_{n-1}, t_c]\!]$ and $[\![s_{n-1}, ..., s_{2n-2}, t_b]\!]$, respectively, and REUSE finds them using a bound as low as $k_{min}^{\text{REUSE}} = n$.

Instead, CLASSIC seeks $k+1$ k-paths, the first one, π, where some $\pi[j]$ satisfies c, and $\pi[0], ..., \pi[j-1]$ satisfy $\mathsf{E}(a\mathsf{U}b)$, as shown by the j k-paths $\sigma_0, ..., \sigma_{j-1}$ having initial state $\pi[0], ..., \pi[j-1]$, respectively (the remaining $k-j$ paths can be any valid k-paths). For the model in Figure 4.9, path π corresponds to $[\![s_0, ..., s_{n-1}, t_c]\!]$, the same as for REUSE; however, path σ_0 cannot be built unless $k \geq 2n-1$, thus $k_{min}^{\text{CLASSIC}} = 2n-1$. □

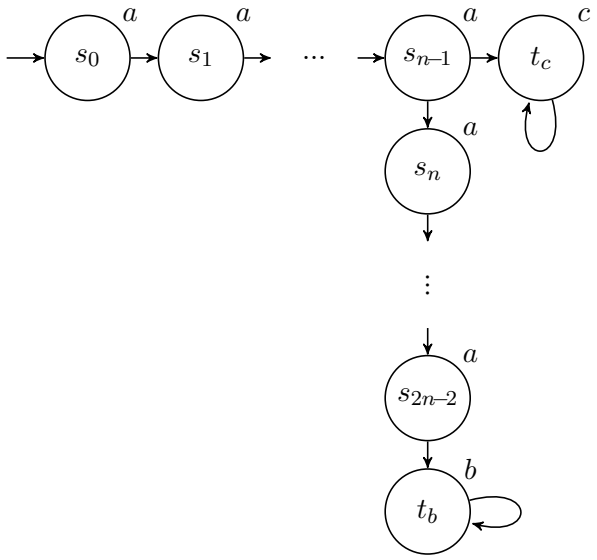


Figure 4.9: A model where $\mathsf{E}(\mathsf{E}(a\mathsf{U}b)\mathsf{U}c)$ holds.

4.4.2 Complexity of Propositional Formulas

We now compare the complexity of propositional formulas generated by CLASSIC and REUSE in terms of the number of symbolic states needed to form a witness, $N_{\text{CLASSIC}}(\varphi) = (k+1) \cdot \mathcal{F}_k(\varphi)$ and $N_{\text{REUSE}}(\varphi) = (k+1) \cdot \mathcal{G}_k(\varphi)$. Recall that modern SAT solvers accept CNF formulas as input and a common practice to transform a propositional formula into CNF is Tseitin transformation [77, 93], which introduces additional boolean variables to avoid an exponential growth in the formula size. Therefore, N_{CLASSIC} and N_{REUSE} do not quantitatively depict the size of the search space a SAT

solver explores or the hardness of the formula, but affect the efficiency of CNF transformation, and it seems plausible to compare and use them as a way to assess the hardness of the formula.

We first consider the simple formula $\mathsf{EG}(\mathsf{EF}a)$. Table 4.1 lists the values of $N_{\text{CLASSIC}}(\mathsf{EG}(\mathsf{EF}a))$ and $N_{\text{REUSE}}(\mathsf{EG}(\mathsf{EF}a))$ w.r.t. k. In particular,

$$\begin{aligned}\mathcal{F}_k(\mathsf{EG}(\mathsf{EF}a)) &= k \cdot \mathcal{F}_k(\mathsf{EF}a) + 1 \\ &= k \cdot (\mathcal{F}_k(a) + 1) + 1 \\ &= k + 1\,,\end{aligned}$$

$$\begin{aligned}\mathcal{G}_k(\mathsf{EG}(\mathsf{EF}a)) &= (k-1) \cdot \mathcal{G}_k(\mu(\mathsf{EF}a)) + \mathcal{G}_k(\mathsf{EF}a) + 1 \\ &= (k-1) \cdot \mathcal{G}_k(a) + (\mathcal{G}_k(a) + 1) + 1 \\ &= 2\,.\end{aligned}$$

It can be seen that $N_{\text{CLASSIC}}(\mathsf{EG}(\mathsf{EF}a))$ grows quadratically, while $N_{\text{REUSE}}(\mathsf{EG}(\mathsf{EF}a))$ grows linearly, as k increases. The larger k is, the more significant the difference becomes.

Table 4.1: Comparison of the two translation approaches on $\mathsf{EG}(\mathsf{EF}a)$.

k	$\mathcal{F}_k(\mathsf{EG}(\mathsf{EF}a))$	$N_{\text{CLASSIC}}(\mathsf{EG}(\mathsf{EF}a))$	$\mathcal{G}_k(\mathsf{EG}(\mathsf{EF}a))$	$N_{\text{REUSE}}(\mathsf{EG}(\mathsf{EF}a))$
1	2	4	2	4
2	3	9	2	6
3	4	16	2	8
4	5	25	2	10
5	6	36	2	12

Then, let us investigate a family of more complex formulas. Let a_i be an atomic proposition for $i \in \mathbb{N}$ and consider the family of ECTL formulas $\{\varphi_1, \varphi_2, ...\}$, where $\varphi_1 = \mathsf{E}(a_0 \mathsf{U} a_1)$ and $\varphi_i = \mathsf{E}(\varphi_{i-1} \mathsf{U} a_i)$ for $i \geq 2$. According to Def. 4.2.3, $\mathcal{F}_k(\varphi_1) = 1$ and, for $i \in \{2, ..., n\}$,

$$\mathcal{F}_k(\varphi_i) = k \cdot \mathcal{F}_k(\varphi_{i-1}) + 1\,,$$

which can be rewritten as

$$\frac{\mathcal{F}_k(\varphi_i) + \frac{1}{k-1}}{\mathcal{F}_k(\varphi_{i-1}) + \frac{1}{k-1}} = k\,.$$

This implies that $\mathcal{F}_k(\varphi_1)+\frac{1}{k-1}, \mathcal{F}_k(\varphi_2)+\frac{1}{k-1}, ..., \mathcal{F}_k(\varphi_i)+\frac{1}{k-1}$ is a geometric series with ratio k. Since its first element $\mathcal{F}_k(\varphi_1)+\frac{1}{k-1}$ equals $1+\frac{1}{k-1}=\frac{k}{k-1}$, its i-th element $\mathcal{F}_k(\varphi_i)+\frac{1}{k-1}$ equals $\frac{k^i}{k-1}$, from which we can conclude that

$$\mathcal{F}_k(\varphi_i) = \frac{k^i-1}{k-1}\,.$$

Finally,

$$N_{\textsc{Classic}}(\varphi_i) = (k+1)\cdot\mathcal{F}_k(\varphi_i) = \frac{k+1}{k-1}(k^i-1)\,. \tag{4.1}$$

Now we compute $N_{\textsc{Reuse}}(\varphi_i)$. According to Def. 4.3.2, $\mathcal{G}_k(\varphi_1)=1$, $\mathcal{G}_k(\varphi_2)=2$, and, for $i\in\{3,...,n\}$,

$$\begin{aligned}\mathcal{G}_k(\varphi_i) &= (k-1)\cdot\mathcal{G}_k(\mu(\varphi_{i-1}))+\mathcal{G}_k(\varphi_{i-1})+\mathcal{G}_k(a_i)+1\\ &= (k-1)\cdot\mathcal{G}_k(\varphi_{i-2}\vee a_{i-1})+\mathcal{G}_k(\varphi_{i-1})+1\\ &= (k-1)\cdot\mathcal{G}_k(\varphi_{i-2})+\mathcal{G}_k(\varphi_{i-1})+1\,.\end{aligned}$$

We build two geometric series by rewriting the equation above as

$$\frac{\mathcal{G}_k(\varphi_i)-\frac{1-\sqrt{4k-3}}{2}\mathcal{G}_k(\varphi_{i-1})-\frac{2}{1-\sqrt{4k-3}}}{\mathcal{G}_k(\varphi_{i-1})-\frac{1-\sqrt{4k-3}}{2}\mathcal{G}_k(\varphi_{i-2})-\frac{2}{1-\sqrt{4k-3}}}=\frac{1+\sqrt{4k-3}}{2}$$

and

$$\frac{\mathcal{G}_k(\varphi_i)-\frac{1+\sqrt{4k-3}}{2}\mathcal{G}_k(\varphi_{i-1})-\frac{2}{1+\sqrt{4k-3}}}{\mathcal{G}_k(\varphi_{i-1})-\frac{1+\sqrt{4k-3}}{2}\mathcal{G}_k(\varphi_{i-2})-\frac{2}{1+\sqrt{4k-3}}}=\frac{1-\sqrt{4k-3}}{2}\,.$$

Therefore, the two geometric series have ratio $\frac{1+\sqrt{4k-3}}{2}$ and $\frac{1-\sqrt{4k-3}}{2}$, respectively. Following similar steps to those used to compute $\mathcal{F}_k(\varphi_i)$, we obtain

$$\begin{aligned}\mathcal{G}_k(\varphi_i)-\frac{1-\sqrt{4k-3}}{2}\mathcal{G}_k(\varphi_{i-1})-\frac{2}{1-\sqrt{4k-3}} &= -\frac{\left(1+\sqrt{4k-3}\right)^2}{2\left(1-\sqrt{4k-3}\right)}\cdot\left(\frac{1+\sqrt{4k-3}}{2}\right)^{i-2}\\ &= -\frac{2}{1-\sqrt{4k-3}}\cdot\left(\frac{1+\sqrt{4k-3}}{2}\right)^{i},\end{aligned}$$

$$\begin{aligned}\mathcal{G}_k(\varphi_i)-\frac{1+\sqrt{4k-3}}{2}\mathcal{G}_k(\varphi_{i-1})-\frac{2}{1+\sqrt{4k-3}} &= -\frac{\left(1-\sqrt{4k-3}\right)^2}{2\left(1+\sqrt{4k-3}\right)}\cdot\left(\frac{1-\sqrt{4k-3}}{2}\right)^{i-2}\\ &= -\frac{2}{1+\sqrt{4k-3}}\cdot\left(\frac{1-\sqrt{4k-3}}{2}\right)^{i}.\end{aligned}$$

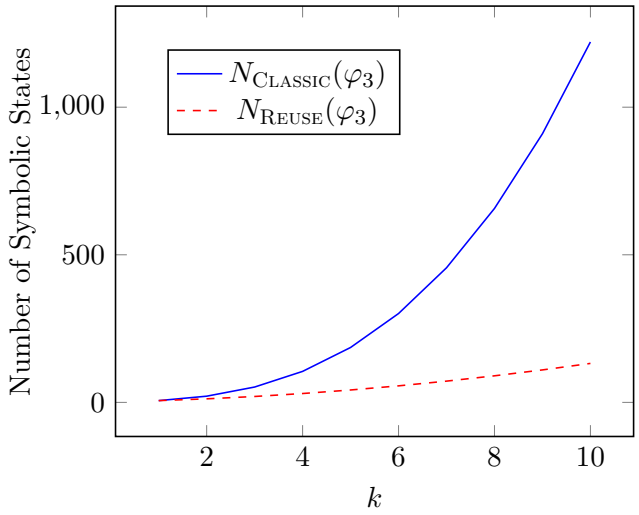


Figure 4.10: Comparison of the growth of $N_{\text{CLASSIC}}(\varphi_3)$ and $N_{\text{REUSE}}(\varphi_3)$.

Combining the two equations above, we have:

$$\mathcal{G}_k(\varphi_i) = \frac{\left(1+\sqrt{4k-3}\right)^{i+2} - \left(1-\sqrt{4k-3}\right)^{i+2}}{(k-1)\cdot 2^{i+2}\cdot\sqrt{4k-3}} - \frac{1}{k-1}.$$

Finally,

$$N_{\text{REUSE}}(\varphi_i) = (k+1)\cdot\mathcal{G}_k(\varphi_i) = \frac{k+1}{k-1}\left(\frac{\left(1+\sqrt{4k-3}\right)^{i+2} - \left(1-\sqrt{4k-3}\right)^{i+2}}{2^{i+2}\cdot\sqrt{4k-3}} - 1\right). \tag{4.2}$$

Note that i is a constant for a given formula in the family we are considering. Therefore, according to Equations 4.1 and 4.2, $N_{\text{CLASSIC}}(\varphi_i) \sim O(k^i)$ and $N_{\text{REUSE}}(\varphi_i) \sim O(k^{\frac{i+1}{2}})$. To visualize the difference between the two, we plot $N_{\text{CLASSIC}}(\varphi_3)$ and $N_{\text{REUSE}}(\varphi_3)$ in Figure 4.10. It can be clearly observed that, as k increases, $N_{\text{REUSE}}(\varphi_3)$ grows significantly slower than $N_{\text{CLASSIC}}(\varphi_3)$.

4.5 Coping with Models Containing Deadlock States

Generally, CTL assumes that the model does not contain any deadlock state. Unfortunately, this assumption is not true for many real-life models. For models containing deadlock states, using either CLASSIC or REUSE may fail to find a witness if deadlock states are part of every witness. Figure 4.11 shows a simple model containing a deadlock state 4. Suppose we are searching a witness for $\mathsf{EF}(a \wedge \mathsf{EF}b)$, which consists of two k-paths. There is no such witness when $k = 1$,

because we must take two steps from the initial state to reach a state where a holds. When $k = 2$, the corresponding propositional formula is also unsatisfiable, because we are not able to build a path of length 2 from state 3.

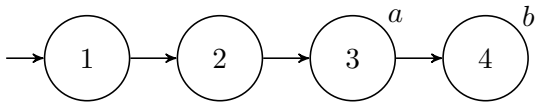


Figure 4.11: A model containing a deadlock state 4.

A common practice to cope with models containing deadlock states is to add a self-loop to every deadlock state. When using a SAT solver, we propose a different approach that adds additional variables to the propositional formula, so that the formula allows k-paths whose actual length is smaller than k. Each symbolic state $\pi_i[j]$ is associated with a boolean variable $\tau_{i,j}$, named *transition flag*, which is **true** if and only if $\pi_i[j]$ is a true successor of $\pi_i[j-1]$, i.e., $\mathcal{N}(\pi_i[j-1], \pi_i[j])$. We modify the encoding of $[M_k]$ as follows:

$$[M_k]_\varphi = I(\pi_0[0]) \wedge \bigwedge_{i=0}^{\mathcal{F}_k(\varphi)-1} \bigwedge_{j=0}^{k-1} (\mathcal{N}(\pi_i[j], \pi_i[j+1]) \vee \overline{\tau_{i,j+1}}) \wedge \bigwedge_{i=0}^{\mathcal{F}_k(\varphi)-1} \bigwedge_{j=0}^{k-1} (\tau_{i,j+1} \Rightarrow \tau_{i,j}).$$

The constraint $\mathcal{N}(\pi_i[j], \pi_i[j+1]) \vee \overline{\tau_{i,j+1}}$ relaxes the assumption that $\pi_i[j]$ must have successors, by simply setting $\tau_{i,l}$ to *false* for $l \in \{j+1, ..., k\}$ if $\pi_i[j]$ is a deadlock state. The constraint $\bigwedge_{i=0}^{\mathcal{F}_k(\varphi)-1} \bigwedge_{j=0}^{k-1} (\tau_{i,j+1} \Rightarrow \tau_{i,j})$ assures that, when $\tau_{i,j+1}$ is **true**, the corresponding symbolic state $\pi_i[j+1]$ is reachable from $\pi_i[0]$, i.e., $\bigwedge_{l=0}^{j} \mathcal{N}(\pi_i[l], \pi_i[l+1])$, otherwise $\pi_i[j+1]$ can take an arbitrary state.

The translation of EX, EF, and EU formulas is also updated (we demonstrate the modification to REUSE; similar modification can also be applied to CLASSIC):

$$[\mathsf{EX}\varphi, s]_k^i = (s \equiv \pi_i[0]) \wedge [\varphi, \pi_i[1]]_k^{i+1} \wedge \tau_{i,1}$$

$$[\mathsf{E}(\varphi\mathsf{U}\rho), s]_k^i = (s \equiv \pi_i[0]) \wedge \Bigg(\left([\rho, \pi_i[0]]_k^{i+1} \wedge \tau_{i,0}\right) \vee \bigvee_{j=1}^{k} \left([\rho, \pi_i[j]]_k^{i+1} \wedge [\varphi, \pi_i[j-1]]_k^{i+1+\mathcal{G}_k(\rho)} \wedge \bigwedge_{t=0}^{j-2} [\mu(\varphi), \pi_i[t]]_k^{i+1+\mathcal{G}_k(\rho)+\mathcal{G}_k(\varphi)+t\cdot\mathcal{G}_k(\mu(\varphi))} \wedge \tau_{i,j}\right) \Bigg)$$

$$[\mathsf{EG}\varphi, s]_k^i = (s \equiv \pi_i[0]) \wedge \bigvee_{j=0}^{k-1} (\pi_i[k] \equiv \pi_i[j]) \wedge [\varphi, \pi_i[k-1]]_k^{i+1} \wedge \bigwedge_{j=0}^{k-2} [\mu(\varphi), \pi_i[j]]_k^{i+1+\mathcal{G}_k(\varphi)+j\cdot\mathcal{G}_k(\mu(\varphi))} \wedge \tau_{i,k}.$$

Of course, if the model is known to be deadlock-free (using a priori knowledge, or some deadlock detection technique), the proposed modification should not be applied, as it increases the size of the resulting formulas.

4.6 Specialization for the Release Operator

Apart from the four temporal operators (X, F, G, and U) we defined in Section 2.2, another temporal operator can be found in some literature:

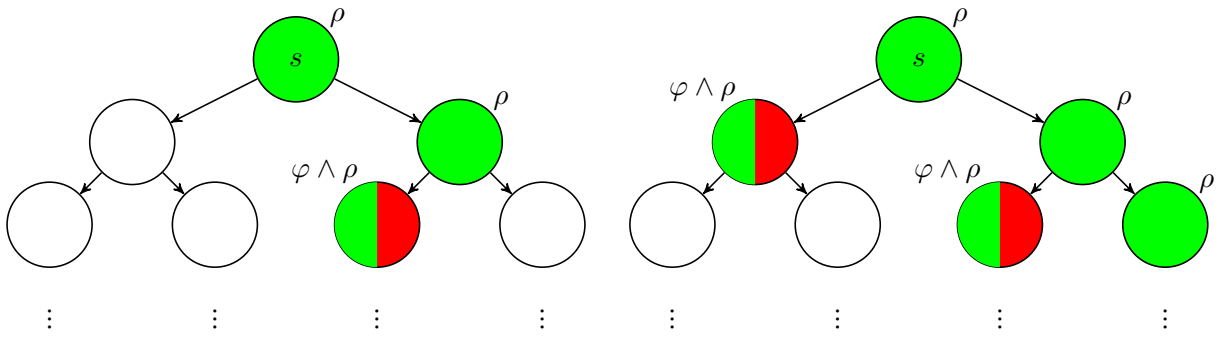
- The R ("*release*") operator is the logic dual of the U operator. It requires that states on the path satisfy the second property until and including a state satisfying the first property. However, the first property is not required to hold eventually. In other words, if there is no state satisfying the first property, every state on the path must satisfy the second property.

Associating the R operator with the two path quantifiers (A and E), we have two additional CTL operators, whose semantics is defined formally as follows:

$$s \models \mathsf{E}(\varphi\mathsf{R}\rho) \Leftrightarrow \exists [\![s_0, s_1, ...]\!] \in Path(s), \forall i \geq 0, s_i \models \rho \vee \exists j \in \{0, ..., i-1\}, s_j \models \varphi$$

$$s \models \mathsf{A}(\varphi\mathsf{R}\rho) \Leftrightarrow \forall [\![s_0, s_1, ...]\!] \in Path(s), \forall i \geq 0, s_i \models \rho \vee \exists j \in \{0, ..., i-1\}, s_j \models \varphi$$

Consider a state $s \in S$. $\mathsf{E}(\varphi \mathsf{R} \rho)$ holds in s if and only if there exists a path $\pi \in Path(s)$ containing a state s' such that $\varphi \wedge \rho$ holds in s' and ρ holds in every state before s' along π, or ρ holds in every state along π. AR is the corresponding universal operators over all the paths from s. These two CTL operators are illustrated in Figure 4.12, where state s is the root of each computation tree.



(a) $s \models \mathsf{E}(\varphi \mathsf{R} \rho)$ (b) $s \models \mathsf{A}(\varphi \mathsf{R} \rho)$

Figure 4.12: The two CTL *release* operators.

ER and AR are dual to AU and EU, respectively:

$$s \models \mathsf{E}(\varphi \mathsf{R} \rho) \Leftrightarrow s \not\models \mathsf{A}(\neg\varphi \mathsf{U} \neg\rho)$$

$$s \models \mathsf{A}(\varphi \mathsf{R} \rho) \Leftrightarrow s \not\models \mathsf{E}(\neg\varphi \mathsf{U} \neg\rho)$$

In particular, ER can be computed using EG and EU:

$$s \models \mathsf{E}(\varphi \mathsf{R} \rho) \Leftrightarrow s \models \mathsf{EG}\rho \vee \mathsf{E}(\rho \mathsf{U}(\varphi \wedge \rho))$$

In symbolic model checking with decision diagrams, ER and AR are often not discussed separately. Sets of states satisfying ER are obtained by computing the union of sets of states satisfying the corresponding EG and EU, while AR is computed based on the duality. There is no known symbolic algorithm computing them directly.

For SAT-based BMC for ECTL we introduced so far, ER formulas are translated into disjunctive formulas of EG and EU. However, with the idea of path reuse, considering ER directly can result in a potentially smaller formula. The main difference is a better translation for the part $\mathsf{E}(\rho \mathsf{U}(\varphi \wedge \rho))$.

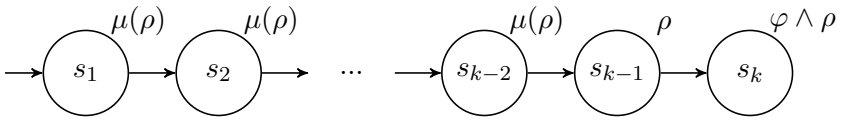
According to the definition of $\mathcal{G}_k$, the number of k-paths to consider a witness for $\mathsf{E}(\rho\mathsf{U}(\varphi \wedge \rho))$ is:

$$\begin{aligned}\mathcal{G}_k(\mathsf{E}(\rho\mathsf{U}(\varphi \wedge \rho))) &= (k-1) \cdot \mathcal{G}_k(\mu(\rho)) + \mathcal{G}_k(\rho) + \mathcal{G}_k(\varphi \wedge \rho) + 1 \\ &= (k-1) \cdot \mathcal{G}_k(\mu(\rho)) + 2\mathcal{G}_k(\rho) + \mathcal{G}_k(\varphi) + 1\end{aligned}$$

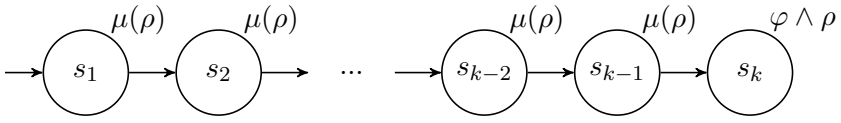
Realizing that $\varphi \wedge \rho$ implies ρ, we may have fewer paths:

$$k \cdot \mathcal{G}_k(\mu(\rho)) + \mathcal{G}_k(\rho) + \mathcal{G}_k(\varphi) + 1$$

Specifically, in the witness that requires all $k+1$ states on the path π to demonstrate $\mathsf{E}(\rho\mathsf{U}(\varphi \wedge \rho))$, the state $\pi[k-1]$ can reuse the subwitness for ρ in the state $\pi[k]$ to build its own subwitness for ρ, therefore we only need to enforce $\mu(\rho)$ in $\pi[k-1]$. Figure 4.13 illustrates such improvement.



(a) With path reuse



(b) With better path reuse

Figure 4.13: Better translation of $\mathsf{E}(\rho\mathsf{U}(\varphi \wedge \rho))$ with path reuse.

The number of paths to consider a witness for $\mathsf{E}(\varphi\mathsf{R}\rho)$ is the larger one between the numbers of paths for $\mathsf{EG}\rho$ and $\mathsf{E}(\rho\mathsf{U}(\varphi \wedge \rho))$:

$$\begin{aligned}\mathcal{G}_k(\mathsf{E}(\varphi\mathsf{R}\rho)) &= max(\mathcal{G}_k(\mathsf{EG}\rho), k \cdot \mathcal{G}_k(\mu(\rho)) + \mathcal{G}_k(\rho) + \mathcal{G}_k(\varphi) + 1) \\ &= max((k-1) \cdot \mathcal{G}_k(\mu(\rho)) + \mathcal{G}_k(\rho) + 1, k \cdot \mathcal{G}_k(\mu(\rho)) + \mathcal{G}_k(\rho) + \mathcal{G}_k(\varphi) + 1) \\ &= k \cdot \mathcal{G}_k(\mu(\rho)) + \mathcal{G}_k(\rho) + \mathcal{G}_k(\varphi) + 1\end{aligned}$$

Compared to the naive approach, the necessary number of paths is $\mathcal{G}_k(\rho) - \mathcal{G}_k(\mu(\rho))$ less. The translation of ER formulas is as follows (without the transition flags):

$$[\mathsf{E}(\varphi \mathsf{R} \rho), s]_k^i = (s \equiv \pi_i[0]) \wedge$$
$$\Bigg(\Big(\bigvee_{j=0}^{k-1} (\pi_i[k] \equiv \pi_i[j]) \wedge [\rho, \pi_i[k-1]]_k^{i+1} \wedge \bigwedge_{j=0}^{k-2} [\mu(\rho), \pi_i[j]]_k^{i+1+\mathcal{G}_k(\rho)+j\cdot\mathcal{G}_k(\mu(\rho))} \Big) \vee$$
$$[\varphi \wedge \rho, \pi_i[0]]_k^{i+1} \vee \bigvee_{j=1}^{k} \Big([\varphi \wedge \rho, \pi_i[j]]_k^{i+1} \wedge \bigwedge_{t=0}^{j-1} [\mu(\rho), \pi_i[t]]_k^{i+1+\mathcal{G}_k(\varphi\wedge\rho)+t\cdot\mathcal{G}_k(\mu(\rho))} \Big) \Bigg)$$

4.7 Experiments

We describe our experimental design in Section 4.7.1 and present the results in Section 4.7.2.

4.7.1 Experimental Design

We implemented both CLASSIC and REUSE in the model checker SMART [19], making use of the SAT solver Nigma [54, 56]. Our benchmark suite is a subset of models and CTL formulas from the CTL cardinality examination of the Model Checking Contest (MCC) 2018 (https://mcc.lip6.fr/2018/). Models are described as Petri nets, most of which have one or more scaling parameters, affecting size and complexity. An experimental instance is a pair of a model instance and an ECTL formula. To select a set of instances eligible for the experiment, we apply the following filtering process:

1. Run SMART up to one hour to determine if the model instance is a safe Petri net (the maximum number of tokens at each place is 1). Discard unsafe Petri nets, so that the places in the remaining model instances can be represented as binary variables (while the places in bounded unsafe Petri nets can be represented using one-hot or binary encoding, we restrict ourselves to safe Petri nets for convenience). In practice, we used the results published in MCC 2018. This leaves 2,256 instances.

2. The formulas associated with the remaining model instances are first simplified if some subformulas can be determined to be **true** or **false**, using knowledge that the model instance

is a safe Petri net. We remove constant **true** or **false** formulas and pure propositional formulas (leaving 1,288 instances). Then, the formulas are transformed into *negation normal form* (NNF) and ACTL formulas are negated to obtain ECTL ones. We remove non-ECTL formulas (leaving 845 instances), non-nested formulas (leaving 278 instances), and formulas that are instances of linear ECTL templates (leaving 110 instances). We also remove formulas containing summations, e.g., the total number of tokens in a set of places (leaving 89 instances). Therefore, the remaining experimental instances have non-trivial ECTL formulas that may have non-linear witnesses and do not contain summations.

Finally, we have 89 instances, taken from 60 model instances from 22 different models. Among the 89 instances, the ECTL formulas hold in 50 instances, do not hold in 32 instances, while it is not known whether they hold in the remaining 7 instances, according to SMART.

To compare the performance of the two translation approaches, we run BMC up to $k = 20$. For each k, the SAT solver is given ten minutes to work on the generated CNF formula. BMC terminates either if a witness is found for some k, or if, for every k up to 20, the SAT solver reports UNSAT or runs out of time. In the latter case, we cannot conclude satisfaction or violation of the corresponding ECTL formula, but we can tell that there is no witness up to the largest k for which the SAT solver reports UNSAT (no "simple" witness exists).

The model instances from MCC 2018 may contain deadlock states, thus we always apply the modification proposed in Section 4.5.

4.7.2 Experimental Results

First, we compare the time spent on transforming propositional formulas into CNF. Since Reuse never generates a larger propositional formula than Classic, it is expected to outperform Classic on this metric, and the results confirm our expectation. Figure 4.14 presents a logscale scatter plot comparing the total time (in seconds) spent on CNF transformation for each experimental instance. A data point above the diagonal means that CNF transformation completes faster using Reuse.

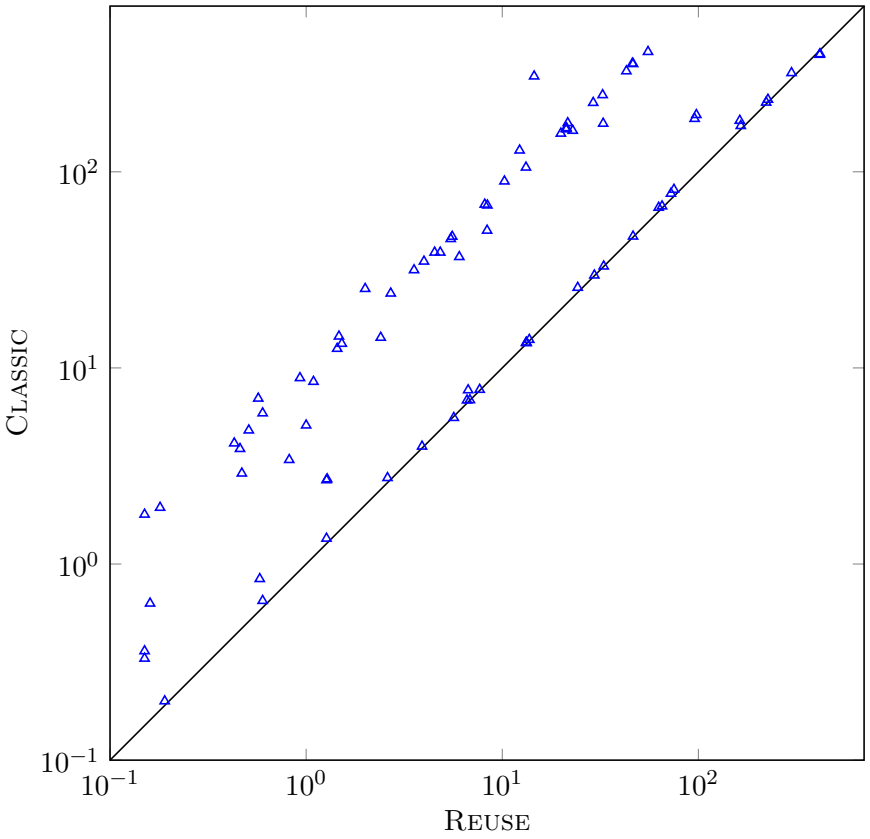


Figure 4.14: Comparing the total time (in seconds) spent on CNF transformation.

From the plot, we can see that Reuse is often substantially better than Classic, and equal to it in the remaining cases, thus we conclude that CNF transformation always benefits from Reuse.

Then, we compare the time spent on satisfiability checking. The results on each experimental instance are classified according to the following categories:

Category 1 (△) The ECTL formula holds, and at least one approach found a witness.

Category 2 (◯) The ECTL formula holds, but neither approach found a witness.

Category 3 (□) The ECTL formula does not hold.

Category 4 (♢) We do not know whether the ECTL formula holds or not.

Figure 4.15 presents a logscale scatter plot comparing the total time (in seconds) spent on satisfiability checking. It also has a zoom-in view of the top-right corner in linear scale for clarity.

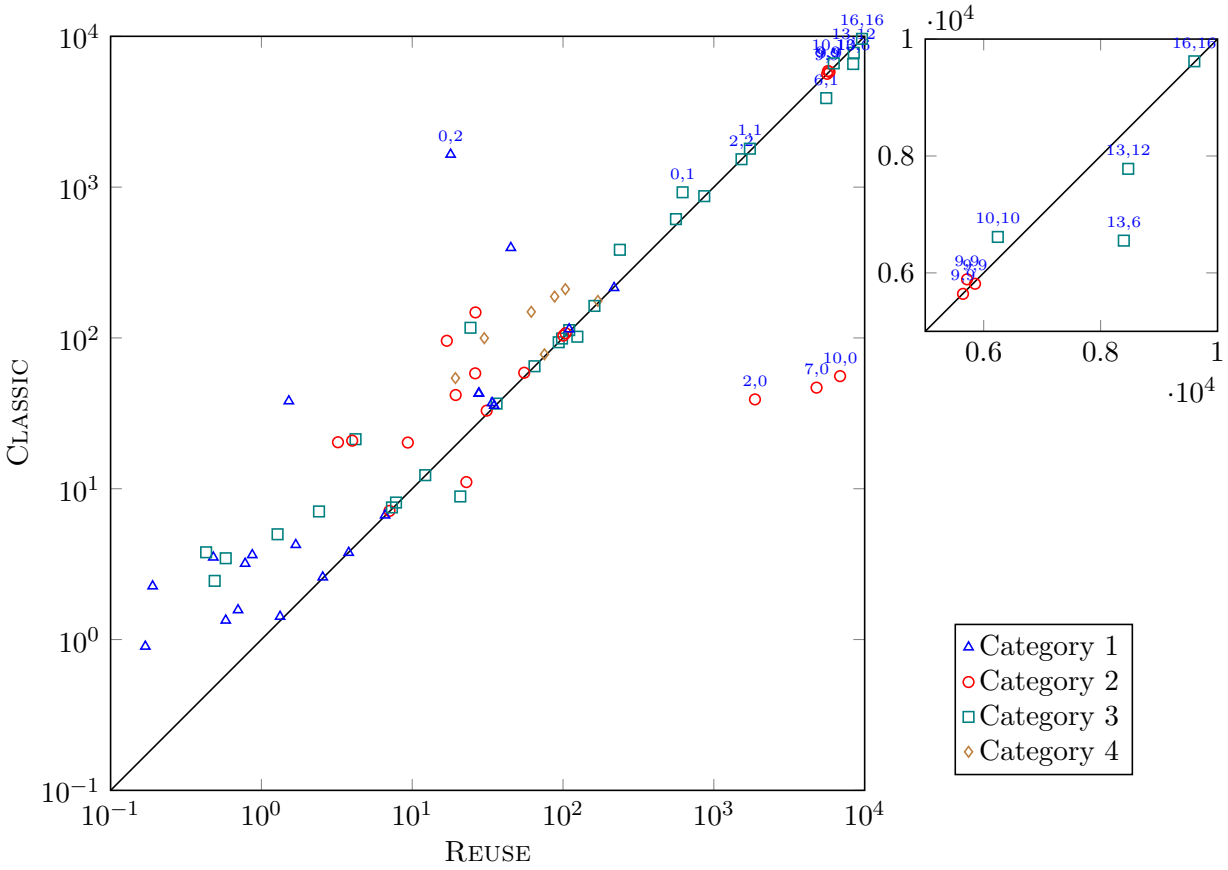


Figure 4.15: Comparing the total time (in seconds) spent on satisfiability checking.

A data point above the diagonal means that the SAT solver terminates (reporting SAT or UNSAT, or running out of time) in a shorter total time, working on the propositional formulas generated by Reuse. The pair of numbers i, j above a data point (omitted if $0, 0$) report the number of timeouts for Reuse and Classic, respectively. For most instances, we can observe a better performance using Reuse.

Instances in Category 1 are those where we can take full advantage of BMC. For them, Reuse always found a witness faster. There is even one instance (the topmost △ in Figure 4.15) where Reuse found a witness but Classic did not. Beside the hardness of formulas, another reason why Reuse is more efficient is that it may find a witness using a smaller k, thus terminate earlier than Classic.

Most instances in Category 2 have relatively "deep" witnesses. BMC may find a witness running with a larger k, at the risk of working on a huge and complex propositional formula. The rightmost three ◯'s below the diagonal are instances where the SAT solver struggled with the formulas generated by REUSE. We noticed more timeouts using REUSE in these instances, which provides more evidence of the well-known fact that there is no strict connection between the size and the hardness of formulas for satisfiability checking [66].

In practice, BMC is not able to answer the instances in Category 3 and Category 4, since the upper bound of the maximal length of symbolic paths is the number of reachable states (see Theorem 4.2.2), often a huge number. In these cases, BMC can only tell us that no "simple" witness exists (up to the largest k for which the SAT solver reports UNSAT). We can see that most of the time, REUSE draws this conclusion faster than CLASSIC.

Finally, we select an experiment instance from Category 1 for a detailed comparison for each value of k in Table 4.2. The model instance is `AutoFlight-PT-05a` and the formula is the negation of $\mathsf{A}\,((p33 \leq p79)\mathsf{U}\,\mathsf{AG}(p89 \leq p88))$. BMC finds a witness for $k = 17$ using CLASSIC and for $k = 13$ using REUSE. **Vars**, **Clauses** and **Literals** are the numbers of variables, clauses, and literals in the CNF formulas. We can see that the CNF formulas generated by REUSE grow slower and are significantly smaller than the ones generated by CLASSIC. **CNF** and **SAT** are the time (in seconds) spent in CNF transformation and satisfiability checking, respectively. CNF transformation always benefits from REUSE, but satisfiability checking may not. With REUSE, the SAT solver spends more time reporting UNSAT for $k = 8$, 9, 10, 11, and 12, though this disadvantage is finally offset by reporting SAT and terminating for a smaller k. An explanation could be that for some model checking problems, a small CNF formula may also have a small unsatisfiable core, which can be deep and hard for a SAT solver to identify. For example, suppose that we are searching a k-path π where $\pi[i] \models_k \mathsf{EF}a$ for any $i \in \{0, ..., k-1\}$. For the formula generated by CLASSIC, the SAT solver reports UNSAT as long as it finds an i such that $\pi[i] \not\models_k \mathsf{EF}a$. However, for the formula generated by REUSE, it reports UNSAT only when it proves that $\pi[k-1] \not\models_k \mathsf{EF}a$.

Table 4.2: Searching for a counterexample to $\mathsf{A}(((p33 \leq p79)\mathsf{U}\,\mathsf{AG}(p89 \leq p88))$ in the model instance `AutoFlight-PT-05a`.

k	Classic					Reuse				
	Vars	**Clauses**	**Literals**	**CNF**	**SAT**	**Vars**	**Clauses**	**Literals**	**CNF**	**SAT**
1	2,357	27,108	56,460	0.04	0.00	2,357	27,108	56,460	0.04	0.00
2	5,384	70,968	147,217	0.12	0.02	4,172	53,661	111,547	0.08	0.01
3	9,353	131,733	272,576	0.23	0.06	5,985	80,213	166,632	0.13	0.02
4	14,266	209,405	432,541	0.41	0.06	7,798	106,766	221,719	0.18	0.03
5	20,123	303,984	627,112	0.59	0.09	9,611	133,320	276,808	0.25	0.05
6	26,924	415,470	856,289	0.82	0.15	11,424	159,875	331,899	0.28	0.06
7	34,669	543,863	1,120,072	1.18	0.22	13,237	186,431	386,992	0.35	0.22
8	43,358	689,163	1,418,461	1.53	0.27	15,050	212,988	442,087	0.40	0.61
9	52,991	851,370	1,751,456	1.68	0.32	16,863	239,546	497,184	0.45	1.67
10	63,568	1,030,484	2,119,057	2.20	0.39	18,676	266,105	552,283	0.52	2.76
11	75,089	1,226,505	2,521,264	2.91	1.64	20,489	292,665	607,384	0.59	5.04
12	87,554	1,439,433	2,958,077	3.03	2.61	22,302	319,226	662,487	0.61	28.66
13	100,963	1,669,268	3,429,496	3.67	2.30	24,115	345,788	717,592	0.64	5.95
14	115,316	1,916,010	3,935,521	4.00	28.05					
15	130,613	2,179,659	4,476,152	5.10	27.50					
16	146,854	2,460,215	5,051,389	5.28	220.14					
17	164,039	2,757,678	5,661,232	6.02	112.17					
Total				38.81	395.99				4.52	45.08

In addition, the SAT solver reports SAT faster on a large formula ($k = 17$ for Classic; $k = 13$ for Reuse) than it reports UNSAT on a small formula ($k = 16$ for Classic; $k = 12$ for Reuse), which confirms that the hardness of satisfiable and unsatisfiable formulas should be evaluated using different criteria.

Next, we evaluate how the specialization for the *release* operator affects the overall performance. Since the CTL formulas from MCC 2018 do not contain the *release* operator explicitly, we negate ACTL formulas containing AU and keep the resulting ECTL ones with $\mathsf{E}(\phi\mathsf{R}\rho)$ where $\mu(\rho) \neq \rho$. Finally, the benchmark consists of 6 instances with ER, taken from different models.

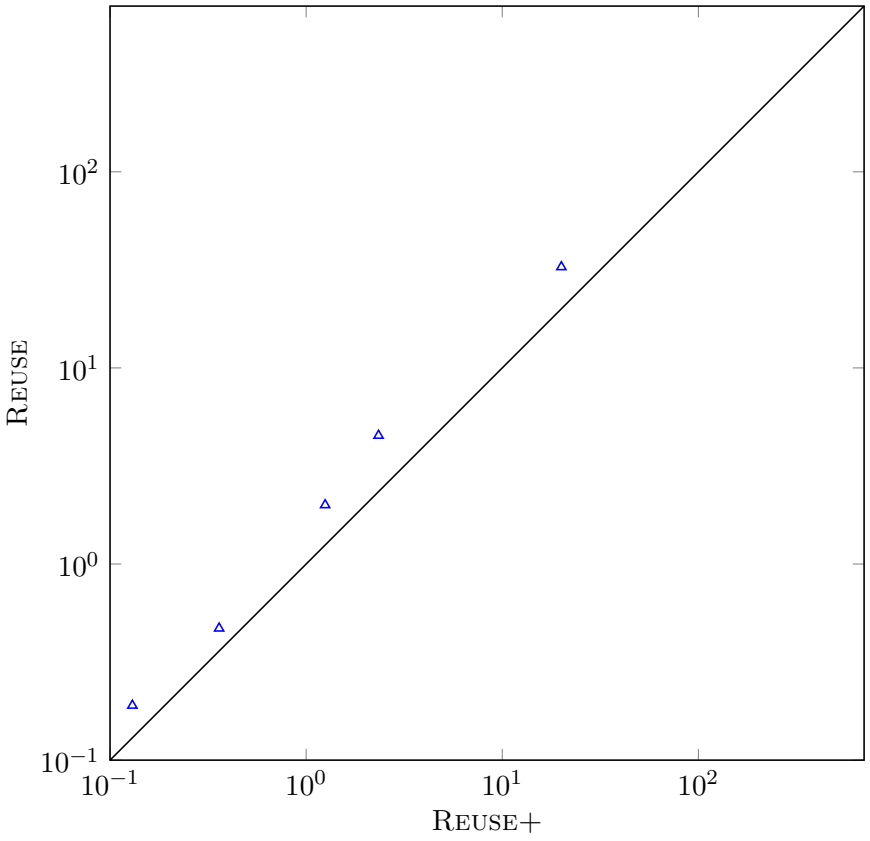


Figure 4.16: Comparing the total time (in seconds) spent on CNF transformation.

Figure 4.16 and Figure 4.17 present logscale scatter plots comparing the total time (in seconds) spent on CNF transformation and satisfiability checking, respectively. REUSE+ corresponds to the experimental instances where we allow the *release* operator in the formula, while REUSE to the ones where the "*release*" semantics is expressed using EG and EU. A data point above the diagonal means that REUSE+ outperforms REUSE in the corresponding phase. One data point is missing in Figure 4.16, and two missing in Figure 4.17, because both approaches have a corresponding runtime that is very close to 0. We can clearly see the performance improvement after specializing for the *release* operator.

In Table 4.3, we select the model instance `AutoFlight-PT-05a` and the negation of the ACTL formula $\mathsf{A}\,((p33 \leq p79)\mathsf{U}\,\mathsf{AG}(p89 \leq p88))$ again for a detailed comparison. In other words, this model instance is actually verified for $\mathsf{EG}(\mathsf{EF}(p89 > p88)) \vee \mathsf{E}((\mathsf{EF}(p89 > p88))\mathsf{U}((p33 > p79) \wedge$

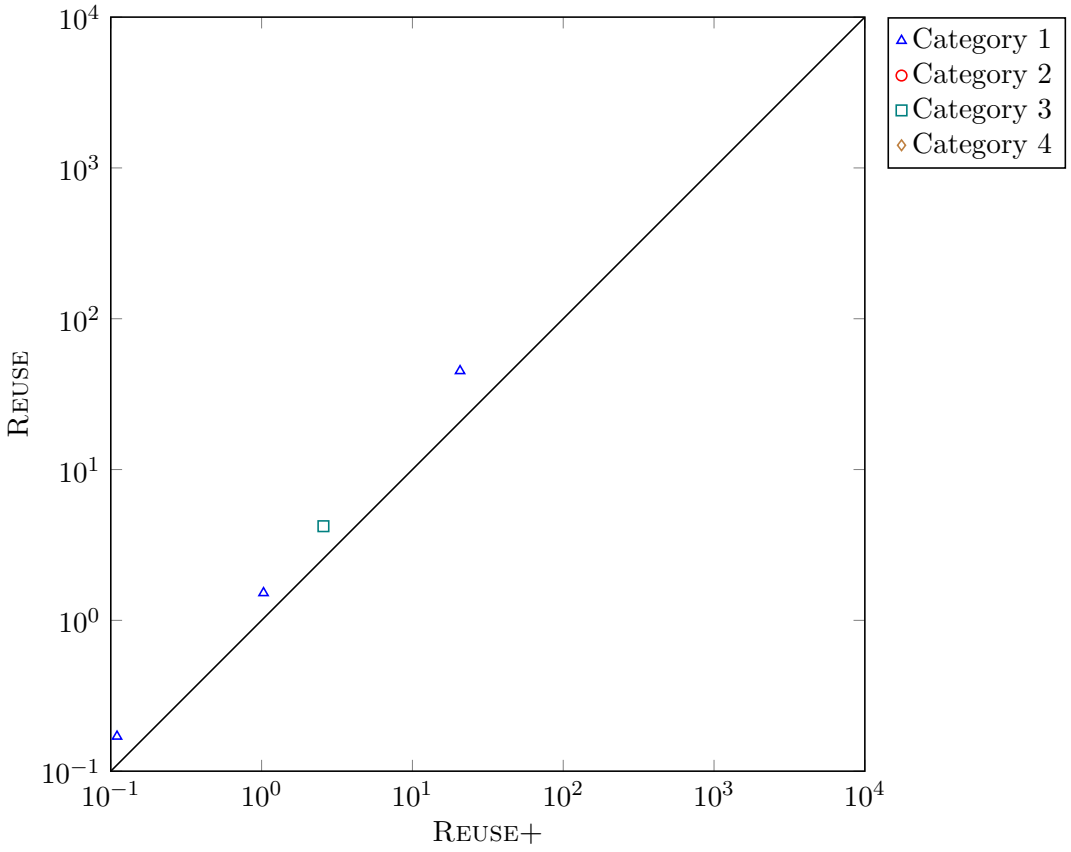


Figure 4.17: Comparing the total time (in seconds) spent on satisfiability checking.

$(\mathsf{EF}(p89 > p88))))$ and $\mathsf{E}((p33 > p79)\mathsf{R}(\mathsf{EF}(p89 > p88)))$, respectively. A witness can be found for $k = 12$ if ER is used in the formula, or for $k = 13$ if the equivalent formula expressed in EG and EU is used instead. For every value of k from 1 to 12, REUSE+ generates a smaller formula, and outperforms REUSE in the runtime of both CNF transformation and satisfiability checking.

It is true that the *release* operator is rarely specified in CTL model checking, which limits the application of its specialization. However, studying the reason of its better efficiency is helpful and inspirational. In fact, given a formula $\mathsf{E}(\phi\mathsf{U}\rho)$, if we can infer that ρ implies ϕ, the translation can be improved with possibly fewer k-paths and produce smaller propositional formulas. We leave the utilization of this observation for future work.

Table 4.3: Searching for a counterexample to $\mathsf{A}\,(((p33 \leq p79)\mathsf{U}\,\mathsf{AG}(p89 \leq p88))$ in the model instance `AutoFlight-PT-05a`.

k	Reuse					Reuse+				
	Vars	**Clauses**	**Literals**	**CNF**	**SAT**	**Vars**	**Clauses**	**Literals**	**CNF**	**SAT**
1	2,357	27,108	56,460	0.04	0.00	1614	18341	38266	0.02	0.00
2	4,172	53,661	111,547	0.08	0.01	2825	36137	75179	0.05	0.01
3	5,985	80,213	166,632	0.13	0.02	4034	53932	112090	0.11	0.01
4	7,798	106,766	221,719	0.18	0.03	5243	71728	149003	0.12	0.02
5	9,611	133,320	276,808	0.25	0.05	6452	89525	185918	0.15	0.03
6	11,424	159,875	331,899	0.28	0.06	7661	107323	222835	0.16	0.04
7	13,237	186,431	386,992	0.35	0.22	8870	125122	259754	0.20	0.11
8	15,050	212,988	442,087	0.40	0.61	10079	142922	296675	0.24	0.35
9	16,863	239,546	497,184	0.45	1.67	11288	160723	333598	0.27	1.15
10	18,676	266,105	552,283	0.52	2.76	12497	178525	370523	0.30	2.65
11	20,489	292,665	607,384	0.59	5.04	13706	196328	407450	0.34	4.14
12	22,302	319,226	662,487	0.61	28.66	14915	214132	444379	0.38	12.19
13	24,115	345,788	717,592	0.64	5.95					
Total				4.52	45.08				2.34	20.70

4.8 Conclusions

We have presented an improved translation to propositional formulas for ECTL BMC, which generates smaller, or at worst the same formulas as the ones generated by the classic approach. Experimental results show that CNF transformation always benefits from our approach, and that satisfiability checking is more efficient most of the time. In addition, we proposed a simple modification to the translation so that it is also defined for models containing deadlock states.

BMC for ACTL formulas having linear counterexamples has been investigated in [96]. It seems promising to combine their work and ours, because our approach has advantages for formulas that are not instantiated from linear templates, thus the two approaches work on disjoint sets of ECTL and ACTL formulas and have no conflict in application.

In Def. 4.3.1, the sufficient predecessor formula of a disjunctive formula does not introduce any improvement: $\mu(\varphi \vee \rho) = \varphi \vee \rho$. However, improvement is possible if we "push down" disjunction.

For example, $\mathsf{E}(a\mathsf{U}b) \vee \mathsf{E}(a\mathsf{U}c)$ can be rewritten into $\mathsf{E}(a\mathsf{U}(b \vee c))$, then $\mu(\mathsf{E}(a\mathsf{U}(b \vee c))) = a \vee b \vee c$. Therefore, simplifying and rewriting CTL formulas [11] may help path reuse achieve a broader applicability.

CHAPTER 5. CONCLUSIONS

We cannot emphasize enough the importance of witnesses and counterexamples in model checking. They are simply and straightforward forms to prove an existential specification or falsify a universal specification. As model checking has become a standard and essential technique in the development process of complex and critical systems, witnesses and counterexamples are receiving more and more attention from scientists and engineers, and leading to new techniques [25, 26, 81]. Since the witness for an existential specification is dual to the counterexample for a universal specification, in this dissertation, we investigate witness generation techniques, in the context of ECTL. Our observations and conclusions are also applicable to counterexample generation for ACTL.

We presented an approach to generate the minimum witness for an arbitrary ECTL formula. The basic idea is to compute the minimum witness size functions for the given ECTL formula and its subformulas, stored as EV^+MDDs. Their computation is based on the saturation algorithm, which exploits event locality and has showed clear advantages over traditional breadth-first approaches for state space generation for asynchronous systems. We explained computing the minimum witness size using saturation for EU and EG formulas in detail. In the case of EG, the concept of transitive closure is borrowed from the graph theory to identify loops. With a global view of witness size, our approach generates witnesses with a minimality guarantee. The experimental results showed that our approach can be much more expensive than the one without this guarantee, which is not surprising. However, it filled the gap of minimum witness generation and can have promising applications to reduce engineers' workload if being combined with relatively cheaper witness generation techniques with no minimality guarantee, especially for companies with rich computational resources.

SAT-based bounded model checking is complementary to decision-diagram-based model checking. For many problems, the boolean functions cannot be succinctly encoded as a BDD, while

SAT-based BMC can be more space-efficient and leverage the success of SAT solvers. BMC inherently has the capability to generate witnesses since a witness can be built from the satisfying assignment if the formula encoding the bounded semantics is satisfiable. We have investigated the BMC approach for ECTL, which considers a witness as a set of bounded paths, and presented an improved translation to propositional formulas by reducing the number of bounded paths through path reuse. The formulas generated by our approach can be significantly smaller, or the same size as in the classic approach in the worse case. Experiments were designed to evaluate and compare the performance of our approach and the classic one in the two most time-consuming processes in BMC, i.e., CNF transformation and satisfiability checking. The results showed that CNF transformation always benefit from our approach, and that satisfiability checking is more efficient most of the time. The idea of path reuse is applicable and useful for checking ECTL formulas where non-linear witnesses may exist. It can be easily combined with techniques proposed for ECTL formulas that guarantee linear witnesses if they exist.

Due to the key role of witnesses in verification, we strongly believe in the value of research on witness generation and look forward to seeing new advanced techniques for this area in the future.

BIBLIOGRAPHY

[1] Fadi A Aloul, Igor L Markov, and Karem A Sakallah. FORCE: a fast and easy-to-implement variable-ordering heuristic. In *Proceedings of the 13th ACM Great Lakes symposium on VLSI*, pages 116–119. ACM, 2003.

[2] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*, volume 9, pages 399–404, 2009.

[3] Gilles Audemard and Laurent Simon. Refining restarts strategies for sat and unsat. In *Principles and Practice of Constraint Programming*, pages 118–126. Springer, 2012.

[4] Jochen Bern, Christoph Meinel, and Anna Slobodová. Global rebuilding of OBDDs avoiding memory requirement maxima. In *International Conference on Computer Aided Verification*, pages 4–15. Springer, 1995.

[5] Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.

[6] Armin Biere. CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT entering the SAT Competition 2018. In *Proceedings of SAT Competition 2018 - Solver and Benchmark Descriptions*, pages 13–14. University of Helsinki, 2018.

[7] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.

[8] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58(11):117–148, 2003.

[9] Beate Bollig, Martin Löbbing, and Ingo Wegener. Simulated annealing to improve variable orderings for OBDDs. In *In Int'l Workshop on Logic Synth.* Citeseer, 1995.

[10] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on computers*, 45(9):993–1002, 1996.

[11] Frederik Bønneland, Jakob Dyhr, Peter G Jensen, Mads Johannsen, and Jiří Srba. Simplification of CTL formulae for efficient model checking of Petri nets. In *International Conference on Applications and Theory of Petri Nets and Concurrency*, pages 143–163. Springer, 2018.

[12] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[13] Randal E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE transactions on Computers*, 40(2):205–213, 1991.

[14] Randal E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

[15] Francesco Buccafurri, Thomas Eiter, Georg Gottlob, and Nicola Leone. On ACTL formulas having linear counterexamples. *Journal of Computer and System Sciences*, 62(3):463–515, 2001.

[16] Jerry R Burch, Edmund M Clarke, and David E Long. *Symbolic model checking with partitioned transition relations*. Carnegie-Mellon University. Department of Computer Science, 1991.

[17] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and computation*, 98(2):142–170, 1992.

[18] Wei Chen and Wenhui Zhang. Bounded model checking of ACTL formulae. In *Theoretical Aspects of Software Engineering, 2009. TASE 2009. Third IEEE International Symposium on*, pages 90–99. IEEE, 2009.

[19] Gianfranco Ciardo, Robert L Jones, Andrew S Miner, and Radu Siminiceanu. Logical and stochastic modeling with smart. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 78–97. Springer, 2003.

[20] Gianfranco Ciardo, Gerald Lüttgen, and Andy Jinqing Yu. Improving static variable orders via invariants. In *International Conference on Application and Theory of Petri Nets*, pages 83–103. Springer, 2007.

[21] Gianfranco Ciardo, Robert Marmorstein, and Radu Siminiceanu. The saturation algorithm for symbolic state-space exploration. *International Journal on Software Tools for Technology Transfer*, 8(1):4, 2006.

[22] Gianfranco Ciardo and Radu Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In *International Conference on Formal Methods in Computer-Aided Design*, pages 256–273. Springer, 2002.

[23] Gianfranco Ciardo and Radu Siminiceanu. Structural symbolic ctl model checking of asynchronous systems. In *International Conference on Computer Aided Verification*, pages 40–53. Springer, 2003.

[24] Gianfranco Ciardo and Andy Jinqing Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 146–161. Springer, 2005.

[25] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[26] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.

[27] Edmund Clarke, Somesh Jha, Yuan Lu, and Helmut Veith. Tree-like counterexamples in model checking. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 19–29. IEEE, 2002.

[28] Edmund Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and complexity of bounded model checking. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 85–96. Springer, 2004.

[29] Edmund Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Computational challenges in bounded model checking. *International Journal on Software Tools for Technology Transfer*, 7(2):174–183, 2005.

[30] Edmund Clarke and Helmut Veith. Counterexamples revisited: Principles, algorithms, applications. In *Verification: Theory and Practice*, pages 208–224. Springer, 2003.

[31] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[32] Edmund M Clarke, Orna Grumberg, Kenneth L McMillan, and Xudong Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Design Automation, 1995. DAC'95. 32nd Conference on*, pages 427–432. IEEE, 1995.

[33] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[34] Dániel Darvas, András Vörös, and Tamás Bartha. Improving saturation-based bounded model checking. *ACTA CYBERNETICA-SZEGED*, 22(3):573–589, 2016.

[35] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[36] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *International conference on theory and applications of satisfiability testing*, pages 61–75. Springer, 2005.

[37] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[38] E Allen Emerson and Edmund M Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *International Colloquium on Automata, Languages, and Programming*, pages 169–181. Springer, 1980.

[39] E Allen Emerson and Edmund M Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer programming*, 2(3):241–266, 1982.

[40] Masahiro Fujita, Hisanori Fujisawa, and Nobuaki Kawato. Evaluation and improvement of boolean comparison method based on binary decision diagrams. In *1988 IEEE International Conference on Computer-Aided Design*, pages 2–5. IEEE, 1988.

[41] Giulio Garbi and Andrew Miner. Decision diagrams for petri nets: A comparison of variable ordering algorithms. *Transactions on Petri Nets and Other Models of Concurrency XIII*, 11090:73, 2019.

[42] Michael JC Gordon and Tom F Melham. Introduction to HOL: A theorem proving environment for higher order logic. 1993.

[43] Thomas A Henzinger, Orna Kupferman, and Shaz Qadeer. From pre-historic to post-modern symbolic model checking. In *International Conference on Computer Aided Verification*, pages 195–206. Springer, 1998.

[44] Andrei Horbach, Thomas Bartsch, and Dirk Briskorn. Using a SAT-solver to schedule sports leagues. *Journal of Scheduling*, 15(1):117–125, 2012.

[45] Jinbo Huang et al. The effect of restarts on the efficiency of clause learning. In *IJCAI*, volume 7, pages 2318–2323, 2007.

[46] Hiroaki Iwashita, Tsuneo Nakata, and Fumiyasu Hirose. CTL model checking based on forward state traversal. In *Proceedings of International Conference on Computer Aided Design*, pages 82–87. IEEE, 1996.

[47] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 14–25. ACM, 2000.

[48] Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 129–144. Springer, 2010.

[49] Matti Järvisalo, Marijn JH Heule, and Armin Biere. Inprocessing rules. In *International Joint Conference on Automated Reasoning*, pages 355–370. Springer, 2012.

[50] Chuan Jiang. Efficient satisfiability solver. Master's thesis, Master thesis, Iowa State University, 2014.

[51] Chuan Jiang, Junaid Babar, Gianfranco Ciardo, Andrew S Miner, and Benjamin Smith. Variable reordering in binary decision diagrams. In *26th International Workshop on Logic & Synthesis*, 2017.

[52] Chuan Jiang and Gianfranco Ciardo. Generation of minimum tree-like witnesses for existential CTL. Figshare (2018), https://doi.org/10.6084/m9.figshare.5926555.

[53] Chuan Jiang and Gianfranco Ciardo. Nigma 1.2. In *Proceedings of SAT Competition*, page 53, 2014.

[54] Chuan Jiang and Gianfranco Ciardo. Nigma 1.2. In *SAT Race*, 2015.

[55] Chuan Jiang and Ting Zhang. Nigma: A partial backtracking sat solver. *Proceedings of SAT Competition*, pages 62–63, 2013.

[56] Chuan Jiang and Ting Zhang. Partial backtracking in CDCL solvers. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 490–502. Springer, 2013.

[57] Timothy Kam, Tiziano Villa, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.

[58] Sujatha Kashyap and Vijay K Garg. Producing short counterexamples using "crucial events". In *International Conference on Computer Aided Verification*, pages 491–503. Springer, 2008.

[59] Henry A Kautz, Bart Selman, et al. Planning as satisfiability. In *ECAI*, volume 92, pages 359–363. Citeseer, 1992.

[60] Saul A Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[61] Timo Latvala, Armin Biere, Keijo Heljanko, and Tommi Junttila. Simple bounded ltl model checking. In *International Conference on Formal Methods in Computer-Aided Design*, pages 186–200. Springer, 2004.

[62] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 123–140. Springer, 2016.

[63] Inês Lynce and João Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. In *Tools with Artificial Intelligence, 2003. Proceedings. 15th IEEE International Conference on*, pages 105–110. IEEE, 2003.

[64] Madjid Maidi. The common fragment of CTL and LTL. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 643–652. IEEE, 2000.

[65] Sharad Malik, Albert R Wang, Robert K Brayton, and Alberto Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *1988 IEEE International Conference on Computer-Aided Design*, pages 6–9. IEEE, 1988.

[66] Zoltán Ádám Mann. Typical-case complexity and the SAT competitions. In Daniel Le Berre, editor, *POS-14. Fifth Pragmatics of SAT workshop*, volume 27 of *EPiC Series in Computing*, pages 72–87. EasyChair, 2014.

[67] Kenneth L McMillan. *Symbolic model checking: an approach to the state space explosion problem*. PhD thesis, PhD thesis, Carnegie Mellon University, 1992.

[68] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th international Design Automation Conference*, pages 272–277. ACM, 1993.

[69] Shin-ichi Minato. Zero-suppressed BDDs and their applications. *International Journal on Software Tools for Technology Transfer*, 3(2):156–170, 2001.

[70] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[71] Rotem Oshman and Orna Grumberg. A new approach to bounded model checking for branching time logics. In *International Symposium on Automated Technology for Verification and Analysis*, pages 410–424. Springer, 2007.

[72] Sam Owre, John M Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.

[73] Lawrence C Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[74] Wojciech Penczek, Bożena Woźna, and Andrzej Zbrzezny. Bounded model checking for the universal fragment of CTL. *Fundamenta Informaticae*, 51(1-2):135–156, 2002.

[75] Carl Adam Petri and Wolfgang Reisig. Petri net. *Scholarpedia*, (4):6477, 2008.

[76] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. *SAT*, 4501:294–299, 2007.

[77] David A Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

[78] Kavita Ravi, Roderick Bloem, and Fabio Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In *International Conference on Formal Methods in Computer-Aided Design*, pages 162–179. Springer, 2000.

[79] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 42–47. IEEE Computer Society Press, 1993.

[80] Petr Savickỳ and Ingo Wegener. Efficient algorithms for the transformation between different types of binary decision diagrams. *Acta Informatica*, 34(4):245–256, 1997.

[81] Natarajan Shankar and Maria Sorea. Counterexample-driven model checking. *SRI International, Menlo Park, CA*, 94025, 2003.

[82] João P Marques Silva and Karem A Sakallah. GRASP—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, pages 220–227. IEEE, 1996.

[83] João P Marques Silva and Karem A Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[84] Radu I Siminiceanu and Gianfranco Ciardo. New metrics for static variable ordering in decision diagrams. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 90–104. Springer, 2006.

[85] Benjamin Smith and Gianfranco Ciardo. SOUPS: a variable ordering metric for the saturation algorithm. In *Proceedings of the 18th international conference on Application of Concurrency to System Design*, 2018.

[86] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 244–257. Springer, 2009.

[87] Paul Stephan, Robert K Brayton, and Alberto L Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):1167–1176, 1996.

[88] Sathiamoorthy Subbarayan and Dhiraj K Pradhan. Niver: Non-increasing variable elimination resolution for preprocessing sat instances. In *International conference on theory and applications of satisfiability testing*, pages 276–291. Springer, 2004.

[89] Jianbin Tan, George S Avrunin, Lori A Clarke, Shlomo Zilberstein, and Stefan Leue. *Heuristic-guided counterexample search in FLAVERS*, volume 29. ACM, 2004.

[90] Seiichiro Tani and Hiroshi Imai. A reordering operation for an ordered binary decision diagram and an extended framework for combinatorics of graphs. In *International Symposium on Algorithms and Computation*, pages 575–583. Springer, 1994.

[91] Alfred Tarski et al. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[92] Elizabeth Tasker. What killed Japan's Hitomi X-Ray satellite? *Scientific American*, 2016.

[93] Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

[94] András Vörös, Dániel Darvas, and Tamás Bartha. Bounded saturation-based CTL model checking. *Proceedings of the Estonian Academy of Sciences*, 62(1):59–70, 2013.

[95] Bow-Yaw Wang. Proving $\forall\mu$-calculus properties with SAT-based model checking. In *International Conference on Formal Techniques for Networked and Distributed Systems*, pages 113–127. Springer, 2005.

[96] Zhaowei Xu and Wenhui Zhang. Linear templates of ACTL formulas with an application to SAT-based verification. *Information Processing Letters*, 127:6–16, 2017.

[97] Andy Jinqing Yu, Gianfranco Ciardo, and Gerald Lüttgen. Bounded reachability checking of asynchronous systems using decision diagrams. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 648–663. Springer, 2007.

[98] Andrzej Zbrzezny. Improving the translation from ECTL to SAT. *Fundamenta Informaticae*, 85(1-4):513–531, 2008.

[99] Hantao Zhang, Dapeng Li, and Haiou Shen. A SAT based scheduler for tournament schedules. In *SAT*, 2004.

[100] Lintao Zhang, Conor F Madigan, Matthew H Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press, 2001.

[101] Yang Zhao and Gianfranco Ciardo. Symbolic ctl model checking of asynchronous systems using constrained saturation. In *International Symposium on Automated Technology for Verification and Analysis*, pages 368–381. Springer, 2009.

[102] Yang Zhao and Gianfranco Ciardo. Symbolic computation of strongly connected components and fair cycles using saturation. *Innovations in Systems and Software Engineering*, 7(2):141–150, 2011.

[103] Yang Zhao, Xiaoqing Jin, and Gianfranco Ciardo. A symbolic algorithm for shortest eg witness generation. In *Theoretical Aspects of Software Engineering (TASE), 2011 Fifth International Symposium on*, pages 68–75. IEEE, 2011.